Integration d'outils pour l'expression et la satisfaction de contraintes dans un generateur de systemes experts

Pierre Berlandier

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél.(1) 39 63 55 11

Rapports de Recherche

N° 924

Programme 1

INTEGRATION D'OUTILS POUR L'EXPRESSION ET LA SATISFACTION DE CONTRAINTES DANS UN GENERATEUR DE SYSTEMES EXPERTS

Pierre BERLANDIER

Novembre 1988

RR_0924

Intégration d'outils pour l'expression et la satisfaction de contraintes dans un générateur de systèmes experts

Pierre Berlandier

INRIA Sophia Antipolis
06560 Valbonne

Travaux effectués dans le cadre du contrat n. 871448 passé par la Direction des Recherches et Etudes Techniques, Direction Scientifique - Section Soutien à la Recherche.

Intégration d'outils pour l'expression et la satisfaction de contraintes dans un générateur de système expert

Résumé

Les langages à base de contraintes apparaissent comme des outils efficaces pour la modélisation, la simulation et la résolution de problèmes. L'intégration d'un système de gestion de contraintes à base de dépendances dans un générateur de systèmes experts renforce la puissance de chacun des deux outils. Néanmoins, les langages de contraintes classiques se révèlent mal adaptés à la gestion d'environnements tels que les bases de faits des systèmes experts. Nous présentons dans ce papier une représentation des contraintes à caractère réflexif. Les méta-contraintes qui en résultent permettent la gestion explicite de l'ensemble des dépendances des contraintes de niveau objet. On conserve ainsi non seulement la cohérence des valeurs du réseau de contraintes mais aussi la cohérence du réseau lui-même vis-à-vis de la base de faits. Cette représentation implique des changements importants sur l'algorithme de propagation de contraintes classique.

Mots clés

Langages de contraintes, propagation de contraintes, systèmes experts, réflexivité.

Integrating Tools for Constraints Expression and Satisfaction in an Expert System Shell

Abstract

Constraint languages have proved to be efficient tools for modeling, simulating and solving problems. Integrating a dependency-based constraint language in an expert system shell results in power and flexibility benefits for both tools. Nevertheless, usual constraint languages turn out to be too weak to deal with such dynamic knowledge bases as expert systems ones. Therefore, we present in this paper a powerful constraint representation based on reflection. The resulting meta-level constraints allow explicit management of the dependency set of object-level constraints so that the dependency network is kept consistent according to the knowledge base evolvement. This representation induces substantial changes on the classic constraint propagation algorithm.

Keywords

Constraint languages, constraint propagation, expert systems, reflection.

Introduction.

" ... when a domain is well understood, it is often possible to describe the objects in the domain in a way that uncovers usefull, interacting constraints..."
Patrick H. Winston

Les outils

La résolution de nombreux problèmes est liée a l'exploration de graphes d'états. Or, dès que l'on sort du cadre d'exemples d'école, cette exploration est rapidement menacée par l'explosion combinatoire. Pour éviter de rencontrer ce monstre redouté, plusieurs types d'outils de l'intelligence artificielle contrôlent la recherche de solutions par des connaissances établies dans le domaine du problème à résoudre. Cette idée a principalement amené le développement des *systèmes experts* dont le raisonnement repose entièrement sur la donnée d'une quantité importante de connaissances.

Par ailleurs, la connaissance que l'on peut avoir sur un domaine s'énonce souvent sous la forme d'un ensemble de contraintes. Trouver une solution d'un problème consiste alors à isoler pour un ensemble de variables caractéristiques, les valeurs qui satisfont simultanément ces contraintes. De nombreux problèmes ont été réduits à des problèmes de satisfaction de contraintes. Parmi ceux-ci, on peut citer l'analyse de circuits [SS80], le filtrage dans l'analyse de scènes [Wal72] et certains aspects des processus de conception et de simulation [Sut63][Bor79]. Des outils adaptés à ce type de problèmes ont été écrits et ont donné naissance à de véritables *langages de contraintes* [Ste80] [Gus86].

Leur coopération

L'aspect déclaratif des règles d'inférence d'un système expert fragilise la cohérence de la base de faits au cours de la résolution d'un problème. En effet, à moins d'imposer un contrôle quasiment impératif sur l'application des règles, il est possible qu'une règle affecte inopinément le raisonnement antérieur du système expert. Certains démons des langages de schémas tels que les "*si-modifié*" permettent des contrôles élémentaires de cohérence. Néanmoins, leurs possibilités restent souvent assez réduites. Les systèmes experts montrent donc un besoin d'outils pour le maintien de la cohérence de leur base de faits au cours de leur raisonnement. C'est pourquoi nous avons établi une coopération entre un générateur de système expert et un langage de contraintes. Ce papier propose l'étude de la réalisation et les implications d'une telle coopération.

1 Satisfaction des contraintes

Au coeur de tout système de contraintes se trouve un mécanisme dont la tâche consiste à trouver un ensemble de valeurs telles que l'ensemble des contraintes exprimées soient satisfaites. Ce chapitre a pour but principal de présenter les détails d'un tel mécanisme: le mécanisme de propagation. Nous y explicitons aussi un algorithme de satisfaction que nous avons développé. Celui-ci reste basé sur le principe de propagation et de là, son mérite principal n'est pas l'originalité. Néanmoins, le manque d'adéquation des algorithmes existants avec les spécifications de notre problème justifie complètement son écriture ainsi que les choix qui le caractérisent.

1.1 Encore un problème NP-complet

Un problème NP-complet est un problème soluble par des algorithmes non-déterministes en temps polynomial c'est à dire des algorithmes qui parcourent parallèlement un nombre arbitraire de chemins qui, s'ils aboutissent, offrent une solution de complexité polynomiale. Pour cette classe de problèmes, tous les algorithmes déterministes connus fonctionnent en temps exponentiel. Ces problèmes, qui impliquent des recherches et des choix, relèvent donc typiquement des techniques de l'intelligence artificielle. [Ric83] va même plus loin en suggérant qu'on peut décrire l'intelligence artificielle comme étant la recherche de techniques permettant d'approcher une complexité polynomiale pour la résolution de problèmes NP-complets.

Le problème qui nous intéresse ici est lui-même NP-complet. On peut en trouver une démonstration dans [Lau86] ou [AHU74]. Ainsi, pour gagner en efficacité, l'algorithme de satisfaction devra perdre en généricité. Les systèmes de contraintes sont donc le plus souvent dédiés à un domaine précis afin de pouvoir intégrer le maximum d'heuristiques et réduire ainsi la complexité de la recherche.

1.2 Satisfaction par génération puis test

Cette section considère le problème de satisfaction sous sa forme la plus générale. Nous noterons $v_1, \ldots, v_n$ les variables du problème, $\mathcal{D}_1, \ldots, \mathcal{D}_n$ leurs domaines respectifs et $|\mathcal{E}|$ la cardinalité d'un ensemble $\mathcal{E}$. Il est important de souligner que les techniques décrites ici ne sont applicables que pour des variables ayant un domaine fini et discret. Elles supposent en effet qu'on peut représenter explicitement ou parcourir en un temps fini l'ensemble des k-uplets du produit cartésien des domaines de k variables quelconques du problème.

En toute généralité, un ensemble de contraintes peut s'exprimer de façon non-constructive[1], à l'aide d'un ensemble de prédicats unaires (de la forme $P_i(v_i)$) ou binaires (de la forme $P_{i,j}(v_i, v_j)$). Il s'agit alors de trouver des valeurs $x_1, \ldots, x_n$ telles que l'on ait:

$$P_1(x_1) \wedge \ldots \wedge P_n(x_n) \wedge P_{12}(x_1, x_2) \wedge P_{13}(x_1, x_3) \ldots \wedge P_{n-1,n}(x_{n-1}, x_n)$$

La stratégie la plus simple et la plus typiquement exponentielle consiste à passer en revue successivement les n-uplets du produit cartésien $\mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ jusqu'à trouver un n-uplet satisfaisant. Il est clair que cette stratégie devient rapidement inutilisable puisqu'on peut tester jusqu'à $\prod_{i=1}^{n} |\mathcal{D}_i|$ combinaisons.

[1] Selon la terminologie de [Gus88], une contrainte *non-constructive* est une contrainte dont on peut seulement tester la validité. [Din86] parle de contrainte *passive*.

Retour arrière chronologique Au lieu de générer une valeur pour chaque variable puis de tester globalement l'ensemble des valeurs obtenues, il apparaît plus astucieux de combiner une instanciation et une évaluation progressive des contraintes du problème. L'ensemble des substitutions possibles pour chaque variable est géré par une pile. Dès que l'évaluation d'une contrainte révèle une contradiction, on remet en cause la valeur de la dernière variable instanciée. On va alors chercher la substitution suivante présente en sommet de pile, puis on reprend l'évaluation. Ce principe d'instanciation progressive avec retour arrière est utilisé par de nombreux systèmes de recherche non-déterministes. Il est notament au centre de la stratégie de recherche du langage PROLOG [CM84]. Le mécanisme est simple à implanter et permet de réduire considérablement l'espace de recherche surtout si la contradiction intervient assez tôt dans le processus d'instanciation. En effet, une contradiction intervenant après la détermination de la k-ième variable élimine l'examen de $\prod_{i \in I} |\mathcal{D}_i|$ combinaisons potentielles, où I est l'ensemble des indices des $(n-k)$ variables non encore instanciées.

Ordonnancement des contraintes Lorsqu'on doit alternativement évaluer des prédicats et instancier les variables intervenant dans ces prédicats, deux stratégies peuvent guider l'enchainement de ces actions :

1. *diriger les instanciations par l'évaluation.* Les prédicats sont alors traités dans un ordre donné et les variables sont instanciées au fur et à mesure qu'on en a besoin pour effectuer les évaluations. C'est le comportement adopté lors de l'évaluation d'une clause PROLOG.
2. *diriger l'évaluation par les instanciations.* Dans ce cas, c'est l'instanciation d'une variable qui va définir le prochain prédicat à évaluer, le but étant de ne pas entrelacer l'évaluation de prédicats dont les ensembles de variables sont disjoints.

Il est clair que dans le cadre du problème de satisfaction de contraintes, l'utilisation de la seconde stratégie permet d'améliorer les performances de la recherche (c.f. [Din86]). Il est par exemple plus judicieux d'évaluer la suite de prédicats

$$P_{i,j}(v_i, vj),$$
$$P_{j,k}(v_j, vk),$$
$$P_{l,m}(v_l, vm).$$

plutôt que

$$P_{i,j}(v_i, vj),$$
$$P_{l,m}(v_l, vm),$$
$$P_{j,k}(v_j, vk).$$

puisque, dans le second cas, une contradiction causée par la valeur de v_j dans $P_{j,k}$ va provoquer $|\mathcal{D}_l| * |\mathcal{D}_m|$ retours arrière avant de remettre en cause la valeur fautive.

Notons que l'ordonnancement de l'évaluation des prédicats se base sur une relation de dépendance implicite entre des ensembles de variables. Cette idée sera raffinée dans la section 1.3.4 où l'introduction de liens de dépendance, mais cette fois-ci au niveau des seules variables, débouchera sur une forme de retour arrière "intelligent".

Contrôle de cohérence des noeuds, des arcs et des chemins Le retour arrière chronologique reste malgré tout une méthode de recherche qui présente souvent un comportement grossier face au problème de satisfaction. Dans [Mac77], trois algorithmes sont proposés qui permettent d'appliquer un pré-traitement sur le domaine des variables et d'isoler de façon efficace certaines valeurs ou combinaisons de valeurs qu'on sait être incompatibles avec l'ensemble des contraintes. Cet ensemble est vu comme un graphe orienté dont les noeuds sont les variables et dont les arcs sont étiquetés par les prédicats. On détecte les incohérences dans ce graphe au niveau des

- **noeuds** On recherche les valeurs qui causent à elles-seules une contradiction au niveau d'un prédicat unaire c'est à dire les $x_i \in \mathcal{D}_i$ tels que $P_i(x_i)$ n'est pas vérifié. Celles-ci peuvent alors être immédiatement éliminées du domaine de la variable.

- **arcs** On suppose que les variables sont instanciées par ordre d'indice croissant. On recherche alors les valeurs $x_i \in \mathcal{D}_i$ telles que $\forall x_j \in \mathcal{D}_j$, $j > i$, $P_{i,j}(x_i, x_j)$ n'est pas vérifié. Ces valeurs constituent des cas pour lesquels le mécanisme de retour arrière va essayer toutes les combinaisons de $\mathcal{D}_{i+1} \times \ldots \times \mathcal{D}_j$ avant de s'apercevoir que x_i est une valeur impossible. On peut donc pareillement les éliminer.

- **chemins** On recherche les couples de valeurs $(x_i, x_j) \in \mathcal{D}_i \times \mathcal{D}_j$ tels que $P_i(x_i)$, $P_j(x_j)$, $P_{i,j}(x_i, x_j)$ sont vérifiés mais $\nexists x_k \in \mathcal{D}_k$ tel que $P_{i,k}(x_i, x_k)$, $P_k(x_k)$ et $P_{k,j}(x_k, x_j)$ soient simultanément vérifiés. Non seulement, comme pour le cas précédent, ces contradictions sont coûteuses à découvrir mais en plus elles peuvent être redécouvertes plusieurs fois. Il est donc important de prévenir la génération de tels couples de valeurs.

L'application de ces algorithmes permet d'éliminer rapidement de nombreuse causes de contradictions mal gérées par le retour arrière chronologique. Cependant, il doit être clair que l'existence de couples de valeurs cohérents au niveau des chemins n'implique pas l'existence d'une solution au problème de satisfaction. Dans [Fre78], les notions de cohérence au niveau des noeuds, des arcs et des chemins sont généralisées à celle de cohérence au niveau de sous-graphes d'ordre k du réseau de contraintes. Il est montré comment cette notion de k-cohérence, exploitée successivement pour k variant de 1 à n, permet d'isoler l'ensemble des n-uplets satisfaisant l'ensemble des contraintes. Ces techniques ont été utilisées avec succès pour des problèmes combinatoires tels le filtrage dans l'analyse de scènes [Wal72] ou la planification temporelle [Rit86].

1.3 Satisfaction par propagation

De nombreux types de contraintes peuvent être exprimés de façon constructive. Les relations associées à ces contraintes sont alors définies soit *extensionnellement* par énumeration de tous leurs k-uplets soit *intentionnellement* par la donnée d'un ensemble d'expressions fonctionnelles représentant chacune de leurs variables. Pour de telles contraintes, la donnée d'un certain nombre de variables peut entrainer la détermination des autres. Ainsi, on peut décrire la contrainte $a = b+c$ par les trois expressions :

$$a \leftarrow b + c$$
$$b \leftarrow a - c$$
$$c \leftarrow a - b$$

Connaissant par exemple les valeurs $a = 3$ et $c = 2$, on déduit naturellement $b = 1$. Cette valeur peut alors être *propagée* c'est à dire, utilisée par toutes les autres contraintes du problème impliquant la variable b. Cette remarque est à la base de l'algorithme de propagation.

1.3.1 Algorithme

Nous présentons dès maintenant l'algorithme dans sa version la plus épurée et dénué de toutes les subtilités qui ne seraient pas directement rattachées au principe de propagation locale.

On dispose d'un ensemble de variables dont les valeurs sont indéterminées et d'un ensemble de contraintes sur ces variables. L'algorithme utilise une pile de contraintes que nous appellerons *Contraintes-Declenchables*. Cette pile est initialement vide. Au fur et à mesure qu'on instancie

des variables du problème, on empile dans *Contraintes-Declenchables* toutes les contraintes attenantes à ces variables. Lorsqu'on décide d'essayer de satisfaire les contraintes, on exécute la boucle suivante:

Algorithme de propagation.

Tant que *Contraintes-Declenchables* n'est pas vide, dépiler le sommet de pile dans *Contrainte-Courante* puis appliquer une des règles suivantes:

1. Si toutes les variables de *Contrainte-Courante* sont déterminées et que la contrainte est déjà satisfaite, alors il n'y a rien à faire.
2. Si toutes les variables de *Contrainte-Courante* sont déterminées et que leurs valeurs contredisent la contrainte, alors il n'est plus nécessaire de continuer la propagation : les contraintes sont dans un état incohérent.
3. S'il n'y a pas assez de variables déterminées pour déclencher *Contrainte-Courante*, alors on ne peut rien faire. Lorsque de nouvelles variables participant à la contrainte seront déterminées, la contrainte reviendra en pile et sera alors reconsidérée.
4. Si *Contrainte-Courante* peut être déclenchée alors, déduire de nouvelles valeurs et empiler dans *Contraintes-Declenchables* toutes les contraintes attenantes aux variables déterminées, sauf bien sur *Contrainte-Courante* et celles qui sont déjà présentes dans la pile.

Si après l'exécution de cette boucle, certaines variables restent sans valeur, le système ne peut alors pas être résolu par seule propagation. Parmi les exemples suivants, nous présentons un tel cas de figure.

1.3.2 Exemples

L'exemple que nous donnons ici est simple et purement artificiel. Il a cependant l'avantage de pouvoir générer de nombreux cas de figure montrant l'éventail des possibilités et des déficiences de l'algorithme de propagation. Il implique quatre variables x, y, z, u et deux contraintes C_1 et C_2 telles que:

$$C_1 : x + y = z \quad C_2 : y + z = u$$

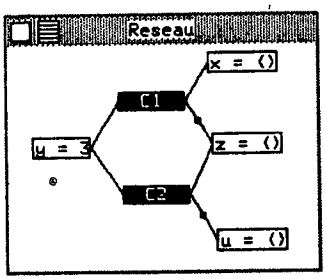
Notons que dans tout ce qui suit, un ensemble de contraintes est représenté par un graphe biparti dont les deux classes de noeuds sont les contraintes figurées en noir d'une part et les variables figurés en blanc d'autre part. Cette représentation, établie dans [FL69] offre une abstraction de la sémantique des contraintes qui permet de se concentrer sur leur aspect topologique et facilite la visualisation du flot de propagation.

1. On suppose que x et y sont connus. Cet exemple va nous permettre de constater que l'ordre dans lequel sont empilées, et donc traitées, les contraintes n'a pas d'importance quand

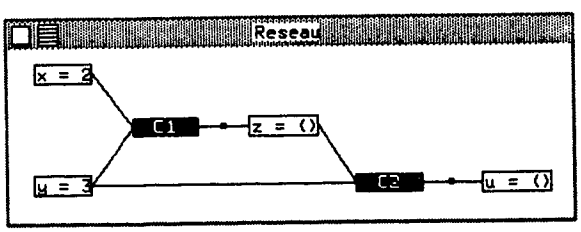
au résultat de la propagation. Avant le lancement de la propagation, la pile *Contraintes-Declenchables* peut en effet valoir $\{C_1, C_2\}$ ou $\{C_2, C_1\}$, selon qu'on a affecté x ou y en premier.

Si la pile vaut initialement $\{C_1, C_2\}$, C_1 est dépilée en premier dans *Contrainte-Courante*. Elle permet de déduire la valeur 5 ($x + y = 3 + 2$) pour la variable z. Les contraintes attenantes à z sont C_1 et C_2. La première est dans *Contrainte-Courante* et la seconde est déjà empilée. On n'empile donc aucune des deux. C_2 est alors dépilée et permet de déduire la valeur 8 pour u. De même que précédemment, C_2, attenante à u n'est pas empilée puisque c'est la contrainte courante. La pile est alors vide et la propagation s'arrête.

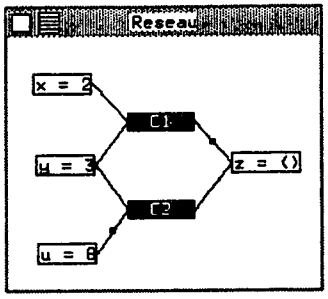
Dans le cas où *Contraintes-Declenchables* $= \{C_2, C_1\}$, C_2 est d'abord dépilée dans *Contrainte-Courante*. Seule la variable y est alors connue. On ne peut donc rien déduire. On passe à la contrainte C_1 qui permet elle de déduire la valeur de z et ré-empile C_2. On reconsidère alors C_2 qui permet à présent de déduire u puisque y et z sont maintenant connus.



2. On connait initialement les valeurs de x, y et u. *Contraintes-Declenchables* vaut au départ $\{C_2, C_1\}$. C_2 permet de déduire la valeur de z. Lorsque C_1 devient la contrainte courante, toutes ses valeurs sont alors connues et la satisfont. Il n'y a donc rien a faire et la propagation s'arrête.



3. Seule est connue la valeur de y. La détermination de cette valeur a empilé initialement dans *Contraintes-Declenchables* les contraintes C_1 et C_2. Il n'y a pas assez de valeurs connues pour déclencher l'une ou l'autre des contraintes. La propagation s'arrête alors sans rien déduire.



1.3.3 Attraits

L'algorithme de propagation offre des attraits largement reconnus puisqu'il est utilisé par une pléthore de systèmes sous une forme ou sous une autre. Parmi ces bons points, [Dav87] dégage les suivants, qui nous semblent particulièrement pertinents :

- L'algorithme est simple à coder et à étendre. Il est facile de comprendre et de suivre ses actions, de les expliquer à l'utilisateur et de les contrôler par un module extérieur intelligent.
- Il se comporte bien vis-à-vis des limitations de temps du calcul qu'on peut lui imposer. En effet, le fait d'interrompre le processus de propagation au cours de son déroulement permet de profiter des inférences qui ont déjà été accomplies.
- La nature locale du comportement des contraintes en fait un paradigme de programmation bien adapté aux traitements parallèles. Chaque contrainte peut être considérée comme un processeur indépendant qui ré-ajuste la valeur de ses arguments lorsqu'un d'entre eux est modifié. Cette approche rappelle les réseaux de processus communiquants présentés dans [KM77]. On trouve aussi dans [Van87] un exemple de parallélisation d'un algorithme de propagation pour un TMS[2].
- L'algorithme convient à des systèmes incrémentaux. En effet, entre deux applications du processus de propagation, on peut ajouter des contraintes dans le réseau. Leur effet est pris en compte dès la propagation suivante.

L'avantage dominant de l'algorithme de propagation reste qu'il permet de ne pas instancier *aveuglément* certaines des variables du problème. Néanmoins, afin de démarrer le processus, il est nécessaire que quelques unes d'entre elles soient arbitrairement valuées. Le choix de ces valeurs peut naturellement conduire à une contradiction après propagation. Il faut alors remettre en cause ces choix. Nous allons voir que ce retour arrière peut, lui aussi, ne pas s'effectuer aveuglément.

1.3.4 Retour arrière dirigé par les dépendances

D'une façon générale, lorsque le processus d'instanciation des variables du problème est dirigé par des inférences sur ces variables, on peut utiliser une forme de retour arrière efficace appelée retour arrière dirigé par les dépendances ou DDB[3] [SS80]. Sa mise en oeuvre implique la présence d'un mécanisme capable de mémoriser le déroulement des inférences accomplies par le

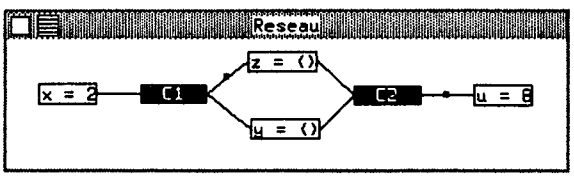
[2]Truth Maintenance System

[3]Dependency Directed Backtracking.

système. Ce système de gestion de dépendances ou RMS[4] construit un graphe de dépendances entre les inférences. Chaque noeud représente le résultat d'une inférence et est relié par un lien de dépendance aux noeuds qui ont permi et amené cette inférence. Ces derniers s'appèllent les *antécédents* du noeud. Le graphe est ensuite utilisé pour tracer les raisons d'une contradiction jusqu'à leur origine. Lorsqu'on rencontre une contradiction, les valeurs fautives qui l'ont impliquée sont retrouvées en remontant aux feuilles du graphe à travers les liens de dépendances (ce qui justifie la qualification "dirigé par les dépendances"). Cette recherche isole un ensemble de noeuds que nous appellerons *prémisses ultimes* du noeud contradictoire. Il faut alors choisir parmi les premisses ultimes le noeud dont on va modifier la valeur. Ce choix fait, on retire les instanciations qui dépendent du noeud choisi (toujours en utilisant le graphe de dépendances) et on relance le processus de recherche. Cette technique s'oppose au retour arrière chronologique qui, lors d'une contradiction, remet indifféremment en cause la dernière instanciation effectuée. Afin d'éviter de reproduire plusieurs fois les mêmes ensembles d'instanciations contradictoires, il faut enregistrer à chaque contradiction l'ensemble des valeurs des prémisses ultimes qui ont provoqué cette contradiction. Le DDB gère donc une liste couramment appelée *no-good-sets*, qui contient tous les k-uplets qui ne doivent pas être reproduits. Cette liste est ensuite consultée lors de chaque instanciation. Il faut remarquer que le fait de n'enregistrer que les k-uplets contradictoires permet d'utiliser le DDB pour des problèmes dont les variables ont des domaines infinis.

Le DDB est un mécanisme largement utilisé par divers systèmes de contraintes tels que ceux décrits dans [Ste80] ou [Mal87]. Il occupe une place centrale dans le fonctionnement des TMS tels que celui de Doyle [Doy79] ou l'ATMS de deKleer [deK86a,deK86b]. Enfin, d'autres utilisations du DDB sont décrites dans [ZMC87] et [Ste87].

1.3.5 Problèmes de circularités



La figure ci-dessus présente un cas où l'ensemble des variables connues est suffisant pour calculer les valeurs de y et z mais où l'algorithme de propagation est incapable d'accomplir une inférence. Ceci est dû à la présence d'une boucle dans le réseau, toutes les variables inconnues appartenant à cette boucle. Dans le cadre de contraintes d'ordre algébrique, il est possible d'offrir des solutions à ce problème de circularité.

Propagation symbolique Une première solution consiste à permettre aux contraintes de propager des expressions symboliques à travers le réseau [SS80]. Ainsi, supposons que l'on donne la valeur symbolique a à la variable y. La propagation de cette valeur à travers C_1 donne $z = a + 2$ puis $y = 8 - (a + 2)$ par C_2. On résout alors algébriquement l'équation $a = 8 - (a + 2)$ puis on substitue le résultat $a = 3$ dans les valeurs de y et z pour trouver $y = 3$ et $z = 5$.

Introduction de points de vue redondants Une autre solution au problème de circularité est de considérer le sous-réseau constitué par la boucle comme une seule et même contrainte et de compléter la définition de cette contrainte par une équation redondante. Dans notre exemple, on

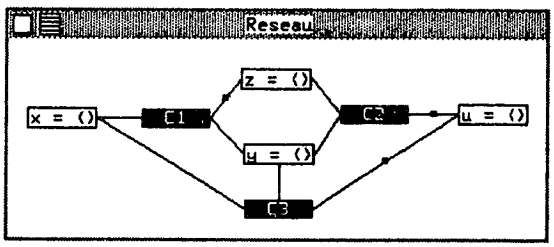
[4]Reason Maintenance System

va donc considérer la contrainte à quatre variables x, y, z et u telle que $x+y=z$ et $y+z=u$. Une simple combinaison de ces deux équations donne par exemple l'équation supplémentaire suivante:

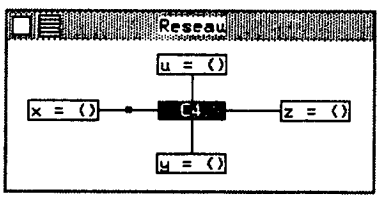
$$x+2*y=u \tag{1.1}$$

Grâce à ces trois équations et quelles que soient les combinaisons de deux variables données, les deux autres peuvent être déduites par propagation. Pour incorporer l'équation 1.1 au réseau, on peut soit :

- ajouter directement au réseau une contrainte correspondant à l'équation comme il est montré sur la figure ci-dessous avec la contrainte C_3. C'est la solution évoquée dans [Ste80] ou [Mal87].



- remplacer le sous-réseau circulaire par une contrainte à quatre variables synthétisant la sémantique des trois équations (c.f. contrainte C_4 sur la figure suivante). Cette technique est appliquée par le système de transformation algébrique de MAGRITTE [Gos83].



On peut remarquer que cette technique correspond en quelque sorte à une compilation dans le réseau de la technique précedente.

Relaxation En dernier ressort, on peut avoir recours à des techniques d'approximation numérique telles que la technique de relaxation. On choisit un ensemble de valeurs initiales pour les variables appartenant à la boucle. L'approximation se fait par itération d'ajustement de ces valeurs. Ces itérations sont contrôlées par une fonction d'erreur sur les valeurs ajustées. Selon le choix des valeurs initiales, ce processus peut diverger ou converger seulement très lentement. Cette lenteur fait que dans des systèmes comme SKETCHPAD [Sut63] ou THINGLAB [Bor79], la relaxation est utilisée si aucune autre solution n'a pu être appliquée.

1.4 Application incrémentale

Nous posons à présent le problème de gérer la cohérence d'un ensemble de variables à travers leur évolution. Cette évolution peut se traduire par l'ajout ou le retrait de variables ou de contraintes ou bien par le changement de la valeur de certaines variables. Ceci s'effectue de façon incrémentale, chaque étape du processus constituant ce que nous appelerons un *état*. L'ensemble des modifications qui sont apportées à un état Θ_i caractérise l'état Θ_{i+1} qui en résulte.

Jusqu'ici, l'algorithme de propagation faisait une partition claire entre les variables dont la valeur était connue et celles qui étaient inconnues. Cette partition nous permettait de reconnaitre quelles étaient les variables à calculer. A présent et si on veut garder un comportement déterministe, le problème va se poser de décider, pour chaque contrainte à satisfaire, laquelle (ou lesquelles) de ses variable *ajuster* en fonction des autres. Ce choix se révèle souvent difficile. Prenons pour exemple simple la contrainte C_1 présentée en 1.3.2 et supposons que dans un état Θ_i on ait $\{x = 2, y = 2, z = 4\}$. Dans l'état Θ_{i+1} suivant, on décide d'affecter $x \leftarrow 3$. Faut-il alors

- modifier y par $y \leftarrow 1$?
- modifier z par $z \leftarrow 5$?
- réinstaller $x = 2$?

Cette décision peut être aidée, de façon locale à chaque contrainte, par différents facteurs. Parmi ceux-ci, on distingue :

la sémantique de la contrainte L'exemple classique d'un choix de modification dirigé par la sémantique est celui du *point médiant* (c.f. [Bor79]) : trois points p_1, p_2 et p_m sont contraint d'être tels que p_m soit le milieu du segment $[p_1,p_2]$. Si on modifie maintenant la position de p_1 (figure 1.1), on a le choix entre bouger le point p_2 (figure 1.3) ou le point p_m (figure 1.2). Il est clair que la seconde solution est plus significative que la première. L'expression de la contrainte sous forme d'une équation lui a donc donné un caractère acausal alors que l'énoncé "p_m *est milieu du segment* $[p_1,p_2]$" distingait clairement un lien de causalité prédominant. Il est donc essentiel qu'outre la sémantique mathématique de la relation qu'elle représente, l'expression d'une contrainte puisse capturer l'existence de ce lien.

la date de modification Il est possible d'étiqueter chaque variable avec sa date de dernière modification. On peut alors choisir d'ajuster la variable la plus anciennement modifiée. On actualise ainsi progressivement l'ensemble des variables. On peut choisir au contraire d'ajuster la variable la plus récemment modifiée, entérinant ainsi la formation d'un lien fonctionnel entre cette variable et les autres partenaires de la contrainte.

l'arbitrage de l'utilisateur Si l'algorithme de satisfaction travaille en mode interactif, on peut faire intervenir l'utilisateur dans le choix de la variable. Néanmoins ce recours doit être très ponctuel et réservé à des cas jugés critiques. Le système risque sinon de devenir rapidement insupportable !

La section suivante s'intéresse à la résolution du problème de satisfaction incrémentale à travers différents systèmes existants. Nous allons voir comment certains de ces systèmes évitent la considération de choix locaux par l'utilisation de l'historique des inférences passées ou la définition d'une heuristique globale.

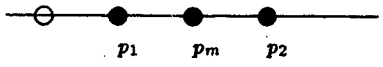


Figure 1.1: On déplace p_1 vers la gauche.

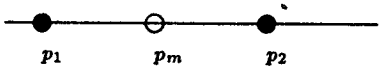


Figure 1.2: Le point milieu p_m s'ajuste pour satisfaire la contrainte.

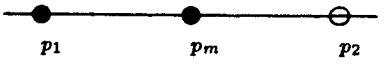


Figure 1.3: Le point p_2 s'ajuste pour satisfaire la contrainte.

1.4.1 Description de systèmes existants

THINGLAB

THINGLAB [Bor79] est un système écrit en SmallTalk. Il offre à l'utilisateur un environnement orienté-objet pour élaborer des simulations. Il permet pour cela de définir, de connecter, de visualiser et de faire évoluer des objets sous contraintes. L'utilisateur de THINGLAB peut attraper à l'aide de sa souris le point p_1 de l'exemple précédent, le déplacer et voir simultanément s'ajuster le point milieu au fur et à mesure du déplacement. Afin de rendre agréables ces interactions, il est essentiel que les techniques de satisfaction employées soient très performantes.

Expression des contraintes Le système permet d'exprimer, de façon statique lors de la définition d'une contrainte, un ordre de préférence quant à la variable à modifier en cas de déclenchement de cette contrainte. L'ordonnancement se fait en accord avec la sémantique de la contrainte. Ainsi, lorsque le processus de satisfaction doit faire un choix, il se réfère à cet ordonnancement pour prendre une décision. THINGLAB permet aussi de contrôler la directionnalité des contraintes exprimées. Pour cela, il faut définir (localement à la contrainte) certaines variables comme étant des *références*, c'est à dire des variables intervenant dans la contrainte mais qui ne pourront pas être modifiées pour la satisfaire.

Satisfaction des contraintes Motivé par un besoin d'efficacité, l'algorithme de satisfaction de THINGLAB présente l'originalité de s'exécuter en deux phases distinctes. La première planifie (sans les effectuer) les ajustements impliqués par les modifications de l'utilisateur et compile une méthode pour effectuer ces ajustements. La seconde phase n'est alors que l'invocation répétée de cette méthode. Par exemple, dès que l'utilisateur déplace un objet, la méthode réalisant la satisfaction des contraintes est générée et compilée à la volée une fois pour toutes. Elle est ensuite utilisée durant tout le déplacement (et pour des déplacements ultérieurs de même ordre).

La phase de planification d'une méthode de satisfaction s'exécute en faisant référence non pas aux valeurs des variables mais uniquement à la topologie du réseau de contraintes. Ceci peut engendrer du travail inutile lorsque, pour une des contraintes visitées, la modification imposée à une de ses variables ne provoque pas de violation de cette contrainte et donc n'implique pas l'ajustement d'une autre variable. Dans un système travaillant de façon incrémentale, on peut tester que la contrainte n'est pas satisfaite avant de poursuivre la propagation et ainsi parcourir un ensemble de noeuds minimal X pour assurer la satisfaction des contraintes. En THINGLAB, la phase de planification ne s'arrête que sur les feuilles du réseau de contraintes. On parcourt alors un ensemble de noeuds $X' \supset X$. Prenons pour exemple les deux contraintes :

$$C_1 : y = Ent(x) \quad C_2 : z = y^2$$

où $Ent(x)$ représente la partie entière de x. Initialement, on a $x = \frac{1}{3}$, $y = 0$ et $z = 0$. Modifions la valeur de x par $x \leftarrow \frac{1}{4}$. Si on propage en testant incrémentalement les contraintes, on s'aperçoit que la modification de x n'implique pas de changer la valeur de y. En revanche, si on procède à une planification sans calcul, on suppose que la modification de x entraine celle de y qui, à son tour, entraine celle de z.

Néanmoins, le surcroît d'opérations engendré par une planification se révèle moins grave qu'il n'y parait puisqu'aucun calcul coûteux n'est effectué sur les noeuds parcourus (on ne recalcule pas de valeurs lors de la planification). Des arguments plus subjectifs sont que des contraintes du type de C_1 sont assez rarement exprimées et qu'il est peu fréquent que les réseaux de contraintes soient d'une profondeur telle que $|X'| \gg |X|$.

CONSTRAINT

Le système décrit dans [Ste80] est assisté d'un RMS (c.f. 1.3.4). Il n'utilise que l'algorithme de propagation étudié en 1.3.1, c'est à dire que seules les variables non-valuées sont calculées. Lorsqu'à partir d'un état stable du réseau on décide de modifier la valeur d'une variable, il faut donc d'abord retirer sa valeur courante. Ce retrait s'effectue à l'aide du graphe de dépendances associé au réseau. On isole d'abord les prémisses ultimes de la variable à modifier. Parmi ces prémisses, on en choisit une. Ce choix se fait soit :

- en prenant la première rencontrée au cours de leur recherche. On peut considérer que c'est la prémisse la plus *directement* responsable de la valeur de la variable à modifier.
- en prenant celle qui implique le moins de répercussions sur les valeurs du réseau. On minimisera ainsi l'ampleur des modifications dans le réseau. Néanmoins, la détermination de cette prémisse peut se révéler couteuse puisqu'elle implique de déterminer l'ensemble de répercussion de chaque prémisse trouvée.
- indépendamment de la structure du réseau, en en choisissant une au hasard ou en demandant l'avis de l'utilisateur.

Une fois la prémisse choisie, on retracte la valeur des variables appartenant à la fermeture transitive de la prémisse selon le graphe de dépendances. On peut ensuite affecter la nouvelle valeur de la variable et lancer le processus de propagation. Nous allons fixer les idées à partir d'un exemple. Soient les deux contraintes

$$\mathcal{C}_1 : z = x + y \quad \mathcal{C}_2 : v = z + u$$

Le réseau initial est celui de la figure 1.4. Toutes les contraintes sont satisfaites. Il est donc stable. Le graphe de dépendance associé qui traduit l'histoire des inférences effectuées est donné par la figure 1.5. On décide de modifier la valeur de v. Pour cela, on recherche l'ensemble des prémisses ultimes de v qui, selon le graphe de dépendances, est $\{x, y, u\}$. Parmi ces prémisses, choisissons par exemple x. On retracte alors la valeur des répercussions de x. Ces repercussions apparaissent en grisé sur la figure 1.6. L'affectation de la nouvelle valeur de v suivie de l'application de la propagation va déduire les nouvelles valeurs de z et x. Il en résulte le nouveau graphe de dépendances montré sur la figure 1.7 où v est maintenant une des prémisses ultimes de x.

Le mécanisme de rétraction que nous venons de présenter n'implique qu'un seul choix par propagation : celui de la prémisse à incriminer. Chacune des prémisses ultimes d'une variable représente une séquence de choix possible sur les variables à modifier. L'ensemble de ces séquences est extrait à partir du graphe de dépendances. Il est donc imposé par la propagation précédente et constitue un sous-ensemble des séquences possibles. Par ailleurs, remarquons que tout comme le système précédent, le fait de planifier l'ensemble des modifications *avant* de les exécuter peut susciter plus de changements que nécessaires.

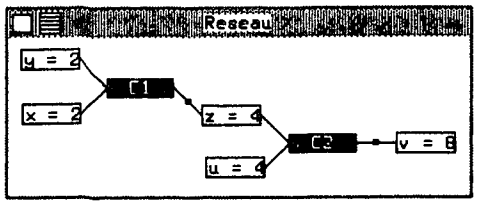


Figure 1.4: Réseau initial.

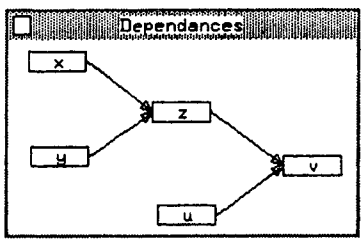


Figure 1.5: Graphe de dépendances initial.

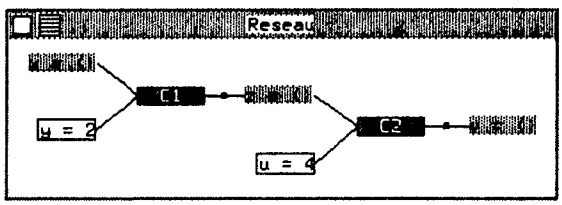


Figure 1.6: Valeurs retirées (en grisé).

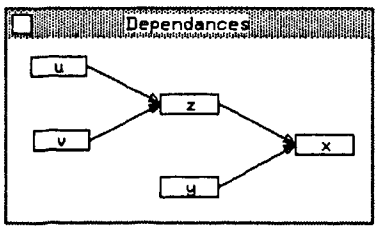


Figure 1.7: Graphe de dépendance aprés propagation.

MAGRITTE

MAGRITTE [Gos83] est un système permettant de définir des figures à l'aide de segments sur lesquels on exprime un ensemble de contraintes. Le système assure le maintien de la cohérence de la figure à la suite de chaque modification qu'elle peut subir. MAGRITTE contourne le problème du choix des variables à ajuster en appliquant la propagation de façon non-déterministe. L'algorithme de satisfaction est alors dirigé par un critère *global* qui est la recherche d'une bonne solution.

Qu'est-ce qu'une "bonne solution" ? c'est l'ensemble *minimum* de modifications qui permet de satisfaire l'ensemble des contraintes. Pour trouver cette solution, l'algorithme effectue une recherche en *largeur d'abord* sur toutes les alternatives de séquences de déclenchement de contraintes qui peuvent être progressivement générées en partant des modifications initiales. Il construit ainsi toutes les séquences de longueur 1, puis toutes celles de longueur 2, etc... jusqu'à trouver la première séquence complète qui satisfasse l'ensemble des contraintes. Ce sera là la solution optimale. Reprenons par exemple le réseau initial de la section précédente (figure 1.4). On décide de changer la valeur de x par $x \leftarrow 3$. La propagation de cette affectation peut engendrer successivement les quatre séquences illustrées par les figures 1.8, 1.9, 1.10 et 1.11 (les variables modifiées apparaissent en grisé). Dans le plus mauvais des cas, MAGRITTE va construire les deux premières séquences et choisir la seconde comme solution puisqu'elle permet de rétablir la cohérence du réseau par une seule modification.

L'inconvénient majeur de cette recherche est qu'elle peut impliquer beaucoup de gaspillage en temps de calcul. La naissance d'une séquence développe, en effet, un calcul propre qui peut être considéré comme un calcul perdu si cette séquence ne se révèle pas être un sous-ensemble de la séquence solution. Ainsi, dans notre exemple, le calcul de z pour la séquence de la figure 1.8 va s'avérer superflu puisque z ne fait pas partie des modifications prescrites par la solution choisie. Ceci n'a pas beaucoup d'importance si on travaille sur des contraintes simples comme des additions ou des multiplications ou si la profondeur du réseau est limitée. En revanche, la perte peut être importante si le rétablissement des contraintes requiert des calculs plus complexes.

Synthèse

En conclusion, il faut retenir :

- Comment la propagation incrémentale *choisit* les variables à ajuster.
  - **Par une stratégie locale :** en chaque contrainte visitée, on choisit quelle sera la prochaine variable modifiée (cas de THINGLAB).
  - **Par une stratégie globale :** on choisit une séquence complète de modifications parmi un ensemble de séquences donné. (cas de CONSTRAINT et de MAGRITTE).
- Comment la propagation incrémentale *effectue* les ajustements.
  - **En une passe :** les ajustements de valeurs sont alors effectués durant le parcours du réseau de contraintes. Cette solution oblige à gérer des environnements si on veut explorer plusieurs solutions (cas de MAGRITTE).
  - **En deux passes :** on planifie d'abord l'ensemble des ajustements et on les effectue ensuite. Il peut alors arriver que la phase de planification explore plus loin que nécessaire (cas de THINGLAB et de CONSTRAINT).

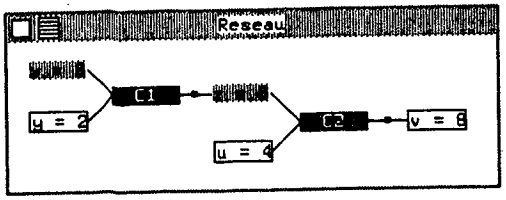


Figure 1.8: Séquence incomplète de longueur 1

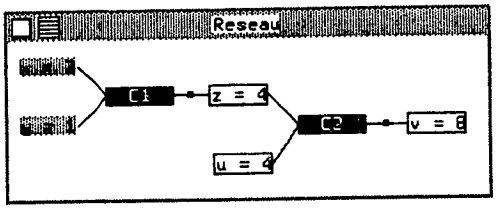


Figure 1.9: Séquence complète de longueur 1

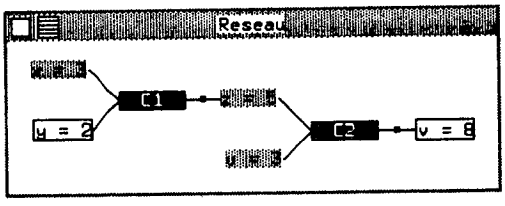


Figure 1.10: séquence complète de longueur 2

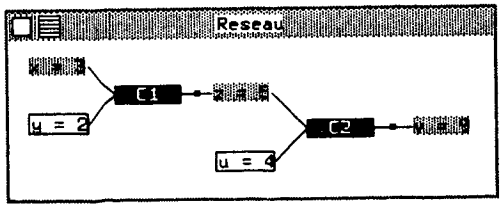


Figure 1.11: Séquence complète de longueur 2

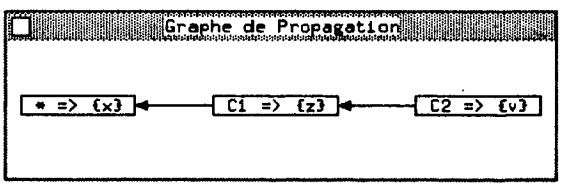
1.4.2 Une propagation plus souple

Le rétablissement local de la cohérence d'une contrainte se fait souvent en ajustant une seule des variables de la contrainte. Nous voulons élargir ce principe en offrant la possibilité de modifier plusieurs valeurs par contrainte déclenchée. [Ba87] montre l'utilité de cette possibilité sur l'exemple de la division entière où la donnée du diviseur et du dividende détermine le quotient *et* le reste. D'autre part, face à une modification, la plupart des algorithmes de propagation cherchent à recalculer immédiatement toutes les valeurs qui doivent être réajustées. Dans certains cas, on peut avoir besoin de ne recalculer qu'un certain nombre de variables qu'on juge intéressantes. [Mal87] donne l'exemple d'une fenêtre visualisant une partie d'un réseau d'objets sous contraintes. Lorsqu'on effectue une modification sur ce réseau, seule nous importe, dans un premier temps, la valeur des variables qui figurent dans la fenêtre. Ceci implique l'existence d'un mécanisme d'évaluation paresseuse des variables ajustées par propagation. Ce mode paresseux devra être utilisé avec précaution car les contradictions apportées par un ensemble de valeurs ne seront plus détectées dès leur introduction [EG87]. Nous décrivons maintenant un algorithme capable d'intégrer les deux caractéristiques précédentes.

Aperçu du mécanisme de propagation

La propagation paresseuse implique une décomposition en deux phases de l'algorithme. Comme en THINGLAB, la première phase va planifier les ajustements en construisant un graphe de propagation. Ce graphe définit les variables à recalculer ainsi que l'ordre de précédence des calculs. La deuxieme phase constitue l'exploitation de ce graphe. On pourra ainsi obtenir la valeur d'une variable modifiée en parcourant le graphe en chaînage arrière à partir de la variable demandée. Notons que, comme pour les méthodes compilées de THINGLAB, l'invocation répétée de la propagation de modifications portant sur un même ensemble de variables peut réutiliser le même graphe de propagation. Il n'y à alors plus qu'à effectuer les calculs.

Les graphes sont constitués de noeuds représentant le recalcul d'un ensemble de variables permettant de satisfaire une certaine contrainte. Un noeud est caractérisé par les attributs *antécedents*, *contrainte*, *conséquents* et *marque*. Ces attributs représentent respectivement l'ensemble des noeuds dont l'évaluation doit précéder celle du noeud lui-même, la contrainte qui effectue les ajustements, l'ensemble des variables à modifier[5] et enfin l'indicateur précisant si le noeud est déjà évalué ou non. Par exemple, la propagation de la modification de x dans le réseau présenté sur la figure 1.4 peut engendrer le graphe (élémentaire) suivant :



Ce graphe signifie que pour restaurer la cohérence du réseau, il faut modifier la valeur de z par C_1 et celle de v par C_2 et que la modification de v doit être postérieure à celle de z.

Un graphe en cours de construction est représenté par une *alternative*. Elle regroupe l'ensemble des noeuds d'un graphe partitionnés en deux catégories : d'un côté, les noeuds à expanser, c'est à dire les noeuds auxquels on doit associer un groupe de variables à modifier et de l'autre les noeuds déjà expansés. Lors de la propagation, les différentes alternatives produites sont rangées

[5] Cet ensemble peut être vide. Cela signifie alors que l'on doit simplement tester la validité de la contrainte associée au noeud.

dans un *sac*. La structure de ce sac est laissée à l'initiative de l'utilisateur. Dans l'implantation courante, ce sac est une *pile*. On réalise par conséquent une construction en profondeur d'abord des alternatives. Il est néanmoins très facile de modifier le module implantant la gestion du sac pour obtenir une construction de type largeur d'abord (en gérant le sac comme une *file*) ou meilleur d'abord (en réordonnant dynamiquement le contenu du sac au fur et à mesure des entrées).

On peut trouver en annexe une description précise de l'algorithme de génération et d'évaluation des alternatives ainsi qu'un exemple déroulant cette génération.

2 Contraintes et système expert

Nous rentrons avec ce chapitre dans le vif du sujet de ce rapport. Nous y traitons en détail de l'intégration d'outils pour l'expression, la création et le maintien de la cohérence de contraintes au sein d'un générateur de systèmes experts. Nous présentons tout d'abord les motivations d'une telle intégration puis nous décrivons précisémment comment nous l'avons implanté. Cette description intervient dans le cadre d'un générateur de systèmes experts particulier. Nous nous efforcerons néanmoins de ne faire appel qu'à des concepts de haut niveau afin de rendre notre approche assez générale pour s'adapter à d'autres générateurs.

2.1 Motivations

La plupart des générateurs de systèmes experts offrent la possibilité de représenter la connaissance de façon mixte. D'une part, l'expressivité des langages à base de schémas permet une représentation aisée de la connaissance structurale sur un domaine. D'autre part, les connaissances de type dynamique qui ne font pas partie de la connaissance sur un schéma particulier sont modélisées par des règles d'inférence. Les systèmes de contraintes offerts dans ces générateurs concernent habituellement le premier type de connaissance, c'est à dire la connaissance structurelle. L'utilisateur peut soit restreindre statiquement le domaine de validité de la valeur de certains attributs ou exprimer des relations entre les attributs d'une même instance de schéma qui seront testées et remises à jour à l'aide d'un mécanisme de démons. Néanmoins, ces mécanismes ne permettent pas d'établir ou de retirer incrémentalement des contraintes entre objets comme cela s'avère souvent nécessaire au cours d'un processus de raisonnement, surtout dans le domaine de la conception. En effet, une grande partie du processus de conception s'attache à la reconnaissance, l'expression et la satisfaction de contraintes [SG87] [EG87]. Les contraintes et leurs interactions sont alors trop nombreuses pour être gérées de façon explicite. Cette tâche peut incomber à un système de contraintes. Nous nous proposons donc d'intégrer un tel système dans un générateur de systèmes experts et d'obtenir une coopération entre les deux parties. De cette coopération, nous attendons principalement des bénéfices au niveau de la facilité et des possibilités d'expression des connaissances, du raisonnement et du maintien de la cohérence de la base de faits du système expert.

2.2 Travaux existants

Les travaux tendant à réunir les fonctionnalités d'un système à base de contraintes avec un autre type de système à base de connaissances semblent être assez rares. Nous en présentons ici deux exemples. [Gus86] présente l'intégration d'un système à base de contraintes nommé CONSAT [GJV87] dans un système de représentation de connaissances hybride. Ce dernier permet d'exprimer des connaissances sous forme de schémas, de règles d'inférence et de clauses PROLOG. La base de faits du système réunit donc des instances de schémas, des objets déduits par les règles et des faits assertés par les clauses PROLOG. Chaque élément communique avec le système de contraintes par l'intermédiaire d'un interprète propre à son formalisme de représentation. Quand un élément est modifié, CONSAT, qui utilise un algorithme de propagation, va réunir les valeurs des éléments appartenant au même réseau que celui-ci. La propagation de la modification va déduire un ensemble d'ajustements qui sera retransmis aux éléments de la base de faits toujours via leurs interprètes respectifs. Chaque formalisme est ainsi responsable de l'interprétation qu'il donne aux valeurs qui lui sont transmises. Par exemple, dans le cas des schémas, si la propagation retourne non

pas une valeur unique mais un ensemble de valeurs, cet ensemble peut être affecté à la facette *valeurs_possibles* de l'attribut modifié. Un des principaux attraits de cette intégration est que l'utilisation d'un système de contraintes évite une grande part de la programmation *explicite* qui serait nécessaire pour maintenir la cohérence de la base de faits.

Un autre exemple d'hybridation est donné dans [Har86]. Le système SOCLE qui y est présenté réunit le système de schémas FRL et le système de contraintes CONSTRAINT décrit en 1.4.1. Il est montré que les contraintes apportent un gain d'expressivité au système à base de schémas en permettant la définition de formules entre des attributs d'instances indépendantes. Les deux composants du système hybride communiquent ici par l'intermédiaire de cellules attachées aux attributs des schémas. Les contraintes sont installées entre ces cellules. Dès qu'une valeur est déterminée par une inférence du système de schéma, elle est répercutée sur la cellule correspondante. Réciproquement, lorsqu'une contrainte détermine la valeur d'une cellule, celle-ci est transmise à son attribut. Lors de la définition des contraintes, les variables sont désignées par des *chemins* à travers les hiérarchies de sous-parties des instances de schémas. On désignera par exemple "la `frequence_de_balayage` du `radar` du `systeme_de_controle`". Un problème survient quand on modifie la valeur d'un des attribut intervenant au milieu d'un tel chemin. Supposons en effet que l'on change le radar du système de contrôle. Il faut alors modifier le réseau en transférant la contrainte de la cellule correspondant à la fréquence de l'ancien radar à la cellule correspondant à la fréquence du nouveau. SOCLE résout le problème en plaçant, lors de la création des contraintes, des attachements procéduraux tout au long du chemin référençant la variable. Ces attachements sont chargés d'effectuer automatiquement le transfert de contrainte lorsque cela est nécessaire. Ce problème doit être considéré chaque fois que les contraintes sont effectivement attachées à une variable et que la variable à été déterminée par une référence susceptible d'évoluer.

2.3 Implantation

L'implantation de notre système a pour hôte le générateur de systèmes experts SMECI[1] [Sme88], développé à l'INRIA et écrit en Le_Lisp [Cha87]. Plutôt qu'une présentation détaillée de ce générateur, la section suivante constitue un dictionnaire qui décrit uniquement l'ensemble des concepts, des fonctionnalités et des mécanismes de SMECI utilisés par le système de contraintes. D'autre part, cette définition de la terminologie employée permettra de faire des rapprochements sans équivoques avec d'autres systèmes.

2.3.1 SMECI

SMECI est un outil qui permet de construire des systèmes experts à représentation mixte de la connaissance: schémas et règles. Son moteur d'inférences construit un arbre d'états qui est exploré à l'aide d'une stratégie standard (*profondeur* ou *largeur d'abord*) ou définie par l'utilisateur selon des critères propres au domaine d'application (*meilleur d'abord*).

Les schémas Le langage de schémas de SMECI possède trois niveaux conceptuels distincts: les *catégories*, les *prototypes* et les *objets*. Nous dirons qu'un objet *est* d'une catégorie et *possède* un prototype.

Categories Les catégories jouent à peu près le rôle des traditionnelles classes des langages objet. Ces classes sont cependant non-extensibles en structure. Il n'est donc pas possible de définir des hiérarchies de catégories.

[1] Système Multi Expert de Conception en Ingéniérie

Une catégorie est décrite par l'ensemble de ses champs. Ces champs sont typés et définissent les propriétés caractéristiques de la catégorie et les liens qu'un de ses objets pourra avoir avec d'autres objets. Tous les objets sur lesquels le système raisonne appartiennent à une catégorie et ils héritent de tous ses champs.

Afin d'éviter de se perdre en conjectures d'ordre technique, nous allons choisir comme base de nos exemples un domaine d'application neutre. On imagine donc que l'on veut gérer l'agencement du mobilier d'un ensemble de bureaux. Le concepteur du système expert voudra créer une catégorie **Meuble**. Un meuble sera par exemple caractérisé par sa hauteur, son prix, son style et le bureau auquel il appartient.

```
Categorie Meuble

   hauteur          type: reel
   prix             type: segment
   style            type: un parmi [Louis-XVI , Directoire , Art-Deco]
   emplacement      type: objet de categorie Bureau
```

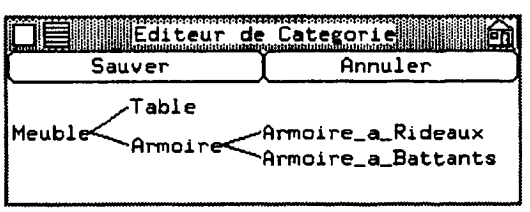
Une autre catégorie que nous manipulerons à travers nos exemples est la catégorie **Bureau** caractérisée par sa surface, l'ensemble des meubles qu'il comprend et par le coût total de cet ameublement.

```
Categorie Bureau

   surface          type: entier
   elements_mobilier type: liste d'objets de categorie Meuble
   cout_mobilier    type: entier
```

En plus des champs décrits par l'utilisateur, une catégorie naît avec la description d'un certain nombre de champs communs à toutes les catégories. Ces champs sont principalement *nom*, *existence* et *prototype*. Ils représentent, pour un objet, son nom qui doit être unique dans une base d'objets donnée, son existence dans la base d'objets courante du système expert et le prototype de cet objet. Nous décrivons ci-dessous ce qu'est un prototype.

Prototypes Les catégories sont raffinées en hiérarchies de sous-classes, non extensibles en structure, appelées prototypes. Cette hiérarchie forme ce qu'on appelle l'*arbre de prototypes* associé à la catégorie. L'arbre de prototypes de la catégorie **Meuble** est, par exemple, le suivant:



Alors que la catégorie décrit la structure des instances, les prototypes permettent de définir de la connaissance associée à la structure, sous forme de valeurs par défaut ou de contraintes de domaine. Ainsi, dans un prototype, on peut spécifier pour chaque attribut de la catégorie:

- l'intervalle de variation des valeurs numériques.

- les listes de valeurs possibles.
- la catégorie des objets, quand le champ relie l'objet à d'autres objets.

Le prototype `Armoire-a-Rideaux`, qui représente un meuble résoluement moderne, peut par exemple définir une restriction sur le prix maximum et les styles possibles.

```
Prototype Armoire-a-Rideaux
   categorie      Meuble
   prix-max       5000
   style          [Art-Deco]
```

Les prototypes permettent également d'effectuer des attachements procéduraux, dépendant de leur position dans l'arbre des prototypes. Un attachement procédural est une méthode que l'on déclenche par l'envoi d'un message à un objet. Tout objet hérite des méthodes attachées aux ancêtres de son prototype, à moins que celui-ci ne les redéfinisse localement.

Les arbres de prototypes sont munis d'un mécanisme automatique de gestion de cohérence des contraintes de domaine, où chaque prototype hérite de l'intersection des contraintes de son père d'une part et de l'union des contraintes de ses descendants d'autre part.

Objets Un objet appartient à une catégorie indiquée à sa création qui définit sa structure. Il possède un prototype qui définit le domaine de variation de chacun de ses champs et les messages auxquels il sait répondre. Les objets sont organisés en réseau par l'intermédiaire de leurs champs de type objet.

Les règles Les règles SMECI s'expriment dans une syntaxe agréable, proche de la langue naturelle et sont compilées en des objets SMECI de catégorie `Regle`. Elles comprennent les deux parties classiques: *prémisses* et *conclusions*.

Premisses Elles décrivent les conditions d'applicabilité de la règle. Elles comprennent une partie *déclaration* dans laquelle on précise, au moyen de variables typées, les objets à instancier puis une suite de prédicats sur les champs de ces objets. Ces prédicats appartiennent à un sous-ensemble de la logique du premier ordre: on peut spécifier des quantificateurs existentiels et universels. On peut par ailleurs utiliser n'importe quel prédicat LISP portant sur les objets ou sur leurs champs

Les prémisses de la règle ci-dessous instancient une armoire à rideaux appartenant au mobilier du *Bureau-102* et dont le prix est supérieur à 1500 francs.

```
defregle cherchez-l-armoire
  soit *b un Bureau
    et *a un Meuble de prototype Armoire-a-Rideaux
          parmi les elements_mobilier de *b
  si le nom de *b = Bureau-102
  et le prix de *a > 1500
alors ...
```

Conclusions Elles permettent de

- créer de nouveaux objets.
- supprimer des objets existants.

- déterminer ou modifier les valeurs des champs des objets instanciés dans la partie prémisses et notamment de
  - resserrer les contraintes réelles sur un objet.
  - préciser davantage le prototype d'un objet, c'est à dire le faire descendre dans la hiérarchie de prototypes.
- effectuer n'importe quelle action en Le-Lisp (impression de message, question à l'utilisateur, appel de routines externes, etc...)

La règle suivante a pour rôle de mettre une armoire disponible dans un bureau qui n'en a pas.

```
defregle attribuer-rangement
  soit *b un Bureau
    et *a une Armoire
  si quelque soit *m un des elements_mobilier de *b
     ona prototype de *m <> Armoire
  et l'emplacement de *a = ()
  alors
  *a est un des elements_mobilier de *b et
  l'emplacement de *a = *b
fin-regle
```

Le raisonnement Le moteur d'inférence de SMECI développe un arbre d'*états* qui représente le raisonnement accompli par le système expert. Cet arbre est parcouru selon une *stratégie* définie par l'utilisateur. Le contrôle de l'application des règles est géré par un interprète de *tâches*.

état Un état est un objet SMECI qui décrit les modifications de la base d'objets induites par l'application d'une règle sur une certaine liste d'instanciations. Ces modifications (ou *transitions*) sont des créations ou des suppressions d'objets et des affectations de valeurs à des champs d'objets. Groupés au sein d'une structure d'arbre, les états figurent le résultat d'un raisonnement. Les feuilles de l'arbre d'états sont des états dits *singuliers*. Ils correspondent à l'arrêt d'une branche de raisonnement. Cet arrêt peut avois plusieurs causes, parmi lesquelles:

- l'état a été reconnu solution du problème par une règle.
- l'état a été reconnu contradictoire par une règle.
- l'état a subi une affectation incohérente (valeur de type erroné ou hors d'un intervalle).

stratégie La stratégie d'un système expert va piloter son raisonnement. C'est un objet SMECI auquel on rattache un certain nombre d'informations qui vont guider la recherche de solutions. Ces informations sont principalement les suivantes :

- Le nombre maximal d'états que pourra créer le système expert au cours de son raisonnement.
- le temps CPU maximal accordé au déroulement du raisonnement.
- Un test de duplication qui permet de reconnaitre des états isomorphes, évitant ainsi de multiplier inutilement les recherches.
- Une fonction d'évaluation qui permet d'attribuer une note à un état dans le cas où on effectue une recherche de type "meilleur d'abord".

tâche L'expression d'un problème à résoudre en SMECI passe par sa décomposition en sous-problèmes indépendants. Ces sous-problèmes sont modélisés par des tâches. Une tâche est un objet SMECI qui est principalement caractérisé par sa *base de règles*. Cette base de règles est un objet décrivant l'ensemble des règles relatives à un sous-problème ainsi que leur utilisation. La structure des tâches permet de les organiser en un graphe qui représente leur enchaînement lors de la résolution du problème.

Un cycle de base du moteur d'inférence se déroule alors schématiquement de la façon suivante:

- Choix du meilleur état parmi les états non explorés.
- Activation de la tâche courante sur cet état. Ceci se traduit par le choix des règles et des instanciations utilisables selon la base de règle associée à la tâche puis par la création de nouveaux états par application des conclusions des règles sur les instanciations sélectionnées.
- Evaluation des états générés suivant la stratégie utilisée.

2.3.2 Expression des contraintes

Afin de pouvoir manipuler nos contraintes de façon dynamique en cours de raisonnement, nous avons étendu la grammaire initiale du langage de règles. La syntaxe de la partie *conclusions* s'est enrichie de la règle < *contraindre* > telle que:

< *contraindre* > ::= `etablir` < *type_de_contrainte* > (< *liste_d_objets* >)

Ce léger sucre syntaxique va permettre une expression purement déclarative des contraintes désirées. De plus, il correspond bien à l'esprit généralement déclaratif des règles d'inférence. La sémantique d'un tel énoncé au sein d'une règle d'inférence est la suivante:

> *Si les premisses de la règle sont valides, alors créer une contrainte liant les champs des objets de* < *liste_d_objets* > *et dont la sémantique est traduite par* < *type_de_contrainte* >.

La règle suivante utilise le type de contrainte **meme-hauteur** pour exprimer que si deux tables sont dans la même pièce, alors, par souci d'uniformité, elles doivent avoir la même hauteur.

```
defregle uniformiser
 soit *t1 un Meuble de prototype Table et
      *t2 un Meuble de prototype Table
 si l'emplacement de *t1 = l'emplacement de *t2
 alors
 etablir meme-hauteur (*t1 *t2)
fin-regle
```

2.3.3 Types de contraintes

Les types de contraintes désignés dans l'énoncé < *contraindre* > sont définis séparément à l'aide de la primitive **defcontrainte**. Cette primitive permet de définir un modèle de contrainte par la donnée d'un prédicat exprimant la relation et de l'ensemble des méthodes utilisables pour satisfaire cette relation. La syntaxe de cette primitive est [2]:

[2]N.B: Certaines < *expression* > admettent un format infixe.

< *defcontrainte* > ::= **defcontrainte** < *nom* >
soit {< *declaration* >}+
predicat < *expression* >
[**methodes** {< *methode* >}+]
fin-contrainte

< *declaration* > ::= < *variable* > { **un** | **une** } < *categorie* >

< *methode* > ::= " [" {< *reference_de_champ* >}+ "] <- " < *expression* >

La sémantique d'une méthode est que, pour satisfaire la contrainte, il suffit d'affecter respectivement les champs d'objets référencés en partie gauche avec les valeurs de la liste rendue par l'évaluation de < *expression* >. Par exemple, pour définir le type **meme-hauteur** qui permettra de maintenir l'égalité entre les hauteurs de deux objets donnés de catégorie **Meuble**, on écrira:

```
defcontrainte meme-hauteur
soit *m1 un Meuble et
     *m2 un Meuble
predicat hauteur de *m1 = hauteur de *m2
methodes [hauteur de *m1] <- hauteur de *m2
         [hauteur de *m2] <- hauteur de *m1
fin-contrainte
```

Les méthodes ci-dessus expriment que si la hauteur d'un des meubles est changée, la hauteur de l'autre doit être affectée de la même valeur. La relation entre **hauteur de *m1** et **hauteur de *m2** est ainsi non-directionelle. Le type défini ci-dessous permettra de contraindre tout objet possédant un champ **hauteur** à conserver cette hauteur dans un intervalle ouvert spécifié.

```
defcontrainte intervalle
soit *o un Objet et
     *borne-inf un entier et
     *borne-sup un entier
predicat hauteur de *o > *borne-inf et
         hauteur de *o < *borne-sup
fin-contrainte
```

On peut noter que ce type ne possède pas de méthodes. Par conséquent, les contraintes qu'il engendrera pourront être testées mais pas rétablies. La compilation de la définition d'un type de contrainte par **defcontrainte** donne naissance à un objet SMECI de catégorie **Relation**. Cette catégorie est propre au système de contraintes. Elle est caractérisée schématiquement par les champs suivants:

```
Categorie Relation

   predicat          type: fonction
   correcteur        type: fonction
```

La sémantique des champs de cette catégorie et l'utilisation qui est faite des objets **Relation** sont expliqués dans la section suivante.

2.3.4 Instances de contraintes

L'introduction de contraintes ne doit pas entrainer l'ajout d'un nouveau formalisme de représentation de connaissance au niveau de l'implantation du noyau. On doit veiller en effet à ne pas alourdir ce dernier et essayer de profiter au maximum des fonctionnalités qu'il offre sur les formalismes existants. Nos contraintes sont donc représentées par des objets standard de catégorie **Relieur**. La description de cette catégorie est la suivante :

```
Categorie Relieur

   dependances        type: liste d'objets de categorie Attribut
   relation           type: objet de categorie Relation
```

- Le champs *dependances* rassemble tous les champs d'objets impliqués par la contrainte. Chaque champ est désigné à l'aide d'un objet de catégorie **Attribut**[3]. La description de cette catégorie est la suivante :

```
Categorie Attribut

   champ              type: symbole
   objet              type: objet de categorie quelconque
   contraintes        type: liste d'objets de categorie Relieur
```

  Le champ *contraintes* d'un attribut contient tous les relieurs dans lesquels cet attribut intervient. Les réseaux de contraintes sont donc modélisés par un graphe biparti dont les noeuds sont alternativement des objets de catégorie **Relieur** et **Attribut**.

- Le champ *relation* pointe vers l'objet de catégorie **Relation** qui définit la relation à maintenir entre les champs de la liste de dépendances. La sémantique d'une relation s'exprime à travers deux champs :
  - Le champ *predicat* désigne une fonction qui, appliqué à la liste de dépendances d'un relieur retourne un booléen qui indique si la contrainte est satisfaite ou non.
  - Le champ *correcteur* désigne une fonction qui, appliquée à un ensemble de dépendances modifiées, retourne une liste de couples (*attribut.valeur*) représentant la liste des ajustements à effectuer pour satisfaire localement la contrainte.

2.3.5 Création des contraintes

L'exécution d'un énoncé < *contraindre* > lors de l'application d'une règle se traduit par :

- la création d'un objet **Relieur**
- la création des objets **Attribut** qui n'existent pas encore
- l'établissement des liens entre attributs et relieur
- l'établissement du lien entre le relieur et sa relation

[3]Cet artifice est inutile quand on travaille dans un langage de schéma dans lequel les champs d'objets sont eux-mêmes des objets

On peut voir la déclaration de contraintes dans les règles comme une macro-définition du langage de règles qui sera expansée en la suite d'actions décrite ci-dessus. Le fait que ces actions se conforment à celles couramment autorisées dans les conclusions des règles garantit d'une certaine façon l'innocuité de la déclaration vis-à-vis du raisonnement du système expert. Le réseau des contraintes s'étend ainsi incrémentalement dans la base d'objets, au fur et à mesure de l'application de règles contraignantes.

2.3.6 Maintien de la cohérence

Après toute création d'un nouvel état par l'application d'une règle, nous devons assurer que celui-ci est cohérent eu égard à son ensemble de contraintes. Nous savons que l'état père de l'état considéré était cohérent puisqu'il a pu générer un fils. Il nous suffit donc de considérer l'ensemble des transitions propres à l'état. On en extrait tous les relieurs créés ainsi que tous les attributs correspond à une des transitions. Ces deux ensembles sont alors passés à l'algorithme de propagation décrit en 1.4.2. Le résultat de la propagation est soit un ensemble d'affectations soit la déclaration d'une contradiction si les modifications apportées par la règle sont incohérentes. Dans le premier cas, les affectations sont effectuées et leur description est ajoutée aux transitions de l'état. Dans le second cas, l'état examiné est déclaré singulier. La branche de raisonnement du système expert s'arrête alors sur cet état.

2.4 Conclusion

La représentation des contraintes dans le formalisme des objets du noyau présente de nombreux avantages. Elle profite notamment du mécanisme de retour arrière dans l'arbre de raisonnement, c'est à dire qu'une contrainte créée dans un état quelconque n'existe plus lorsqu'on remonte sur un de ses état ancêtres. Néanmoins, un problème survient lorsqu'un utilisateur retors vient à essayer de contraindre les champs d'un relieur ou d'une relation. A priori, ceci est tout à fait légal mais il devient alors impossible de garantir l'intégrité de la sémantique des contraintes. Prenons par exemple le cas d'une contrainte C_{meta} portant sur la liste de dépendances d'une autre contrainte C_{objet}. L'algorithme de propagation ne définit pas de précédence entre le calcul des dépendances d'une contrainte et l'utilisation de ces dépendances. Il pourrait donc arriver qu'une des variables de C_{objet} soit recalculée sur la base d'une certaine liste de dépendances puis qu'au cours des ajustements suivants, la valeur de cette liste soit modifiée par le déclenchement de C_{meta}. L'état résultant serait alors incohérent. Une solution simple et radicale pour éviter ces problèmes est d'interdire l'expression de contraintes impliquant des champs d'objets système. Une approche plus délicate consiste à insérer dans l'algorithme de propagation un traitement *ad hoc* pour les champs d'objets système. Dans tous les cas, la gestion de restrictions ou d'exceptions nous semble aller à l'encontre du principe d'uniformité qui guide (ou devrait guider) tout développement de sous-systèmes en SMECI. D'autre part et ainsi que nous l'enseignent les préceptes fondamentaux du génie logiciel, l'implantation de tels cas particuliers implique souvent sur le code une augmentation de sa taille, de sa complexité, le verminage de parties initialement saines et des difficultés de maintenance. Au vu de cette liste imposante de monstruosités, la conclusion qui s'impose est que notre algorithme de propagation doit être refondu afin d'offrir un comportement uniforme et sémantiquement correct quels que soient les objets contraints.

D'autre part, une attitude saine lorqu'on est confronté à des considérations de type "traitement ad hoc" ou "activité méta" au cours de l'élaboration d'un système est de penser à une *architecture réflexive*. L'objet du chapitre suivant est précisément d'exposer l'impact de la réfléxivité sur notre système de contraintes.

3 Vers un système réflexif

Dans ce chapitre, nous abordons le concept de réflexivité appliqué aux systèmes informatiques. Ce concept s'avère assez délicat à appréhender; en particulier, la frontière au delà de laquelle un système mérite d'être qualifié de réflexif est souvent mal perçue. En revanche, il apparait clairement que les approches réflexives permettent de résoudre de façon élégante des problèmes inhérents à l'organisation de l'activité du système. Outre des considérations d'ordre esthétique, nous voulons montrer que la prise en compte du concept de réflexivité lors de l'élaboration d'un système à base de contraintes confère à celui-ci un gain important en matière d'expressivité et de souplesse.

3.1 Généralités sur les systèmes réflexifs

Un système exécute sur un domaine les actions décrites par un programme. Le système arbore un comportement réflexif lorsqu'il se prend lui-même pour domaine d'activité, c'est à dire qu'il applique des actions à une représentation de son état. L'activité réflexive ne contribue donc pas directement à la résolution d'un problème puisqu'elle n'agit pas sur les objets du domaine du problème. En revanche, elle a pour but de faciliter et d'adapter l'activité du système face aux problèmes qui lui sont soumis.

L'intégration du concept de réflexivité passe par la détermination d'un *modèle* pour un certain nombre de composants du système. Cette détermination s'effectue relativement à la théorie que l'on a des éléments du niveau objet [Bat83]. Le modèle va fixer l'ensemble des activités que le système peut accomplir sur lui-même; celui-ci sera plus ou moins adapté à une activité réflexive donnée. Il n'existe donc pas de modèle complet ou optimal. La plupart des systèmes réflexifs n'offrent qu'une seule représentation d'eux-mêmes. [Mae86] pose les bases d'un langage orienté-objet réflexif offrant plusieurs points de vue sur un même programme.

Pour que la réflexivité soit intéressante, il faut qu'il existe un *lien de causalité* entre l'activité du système et sa représentation. Ce lien de causalité garantit que l'activité et sa représentation sont toujours en accord, c'est à dire que la modification de l'une entraine une modification correspondante de l'autre. Par exemple, une jauge graphique sur un écran est associée à une variable représentant la quantité jaugée. Il existe un lien de causalité entre ces deux objets si la modification de la quantité déclenche un réaffichage de la jauge et si une action sur la jauge à l'aide de la souris affecte une nouvelle valeur à la quantité jaugée. Ainsi, à l'issue d'une activité réflexive, la reprise de l'activité de niveau inférieure se trouve effectivement affectée par les modifications de sa représentation. Un système qui offre uniquement la possibilité d'observer son comportement sans pouvoir le modifier ne peut pas être qualifié de réflexif [Mae86].

Une fois le modèle du système déterminé et relié à ce qu'il modélise, il reste encore à déterminer quand et comment utiliser les capacités réflexives du système. A un niveau donné, il faut pouvoir arrêter l'activité du système, passer à un niveau méta puis retourner au niveau objet après avoir manipulé sa représentation. Une architecture réflexive doit donc définir un *contrôle de l'activation du processus réflexif.*

[Fer87] distingue trois type d'approches réflexives selon le type d'activité du système et les connaissances utilisées au cours de son activité réflexive. On trouve ainsi une réflexivité *structurale*, *procédurale* et *conceptuelle*. Nous présentons plus en détail la réflexivité procédurale car notre système de contraintes entre dans ce cadre.

3.1.1 Réflexivité procédurale

Un langage dont les programmes ont la possibilité d'accéder à une représentation d'eux-mêmes et d'intervenir sur le déroulement de leur processus d'exécution est doué d'un comportement réflexif. Par son uniformité de représentation entre les données et les programmes, LISP offre un terrain propice à l'introduction du concept de réflexivité. En effet, LISP permet d'exécuter des données et de manipuler des programmes. D'autre part, la plupart de ses dialectes donnent accès à leur environnement par des fonctions telles que `boundp`[1] ou à leur pile d'exécution par `cstack`[1], `tag`[1] ou `exit`[1]. Ces fonction sont souvent utilisées pour construire des outils de trace ou d'exécution incrémentale (pas-à-pas). Néanmoins, l'environnement d'exécution n'est pas explicitement représenté par des objets du langage.

3-LISP [Smi82] est un dialecte réflexif de LISP dans lequel une fonction réflexive accède à l'environnement d'exécution de la fonction qui l'a invoquée. Cet environnement est représenté par des variables LISP : l'ensemble des variables liées est donné sous forme d'une a-liste et la continuation par une s-expression. Le lien de causalité entre modèle et activité existe naturellement en 3-LISP du fait de l'utilisation d'un interprète méta-circulaire [Cou78]. Un interprète méta-circulaire utilise en effet une représentation explicite de l'interprétation pour accomplir effectivement cette interprétation. Enfin, l'activité réflexive d'un tel système est explicitement contrôlée par les programmes eux-mêmes c'est à dire que le code lui-même comporte des appels aux fonctions réflexives.

3.1.2 Apports des approches réflexives

De nombreuses activités d'un système informatique sont réflexives par essence c'est à dire qu'elles nécessitent l'accès à une représentation du programme ou de son état. Ces activités peuvent ne faire intervenir que des concepts intérieurs au système. C'est le cas, par exemple, des fonctions de trace et d'exécution pas-à-pas. D'autres impliquent l'intervention de connaissances extérieures au système. Parmi celles-ci, on trouve l'apprentissage ou bien le raisonnement sur le contrôle. Ainsi, un système qui apprend doit être capable de s'auto-modifier à la suite d'inférences qu'il vient d'accomplir sur un problème quelconque. Ces modifications font évoluer son comportement ultérieur confronté à un nouvel exemplaire du même problème. Evidemment, il est essentiel de procurer au système un critère permettant de juger le bien-fondé de ces auto-modifications afin qu'il progresse.

[Cor88] dégage certains avantages des approches réflexives du point de vue génie logiciel dans le domaine de l'intelligence artificielle. Ces avantages apparaissent aussi bien du côté du développeur que de l'utilisateur du système. Tout d'abord, la programmation d'un système dans le même formalisme que celui des connaissances peut amener une réduction importante de la taille du code du système. On limite de toutes façons le nombre d'applications utilitaires à écrire comme les gestionnaires d'entrée-sortie, les outils d'édition ou de trace. Dans le même temps, l'utilisateur n'est pas obligé d'assimiler autant de formalismes que de concepts. Un système réflexif apparaît aussi plus facilement extensible ou modifiable. En effet, dès lors que l'on maitrise bien le formalisme de représentation des connaissances, on peut se lancer dans des opérations de chirurgie et d'extension de la méta-connaissance.

3.2 Réflexivité et contraintes

Nous avons vu dans le chapitre précédent que la représentation de nos contraintes permet d'exprimer naturellement des méta-contraintes. Or notre algorithme s'avère incapable de gérer correctement de telles contraintes. On peut alors soit penser que l'algorithme n'est pas approprié, soit

[1] Fonctions du dialecte Le_Lisp [Cha87].

douter du bien fondé de la représentation des contraintes. La section suivante montre clairement sur des exemples l'utilité de méta-contraintes. Le choix de la représentation étant justifié, nous montrons alors l'influence de telles contraintes sur l'algorithme de satisfaction.

3.2.1 Un cas pathologique

Le problème que nous allons décrire se pose lorsqu'on désire représenter des contraintes dans une base d'objets dont l'organisation est extrèmement dynamique. On suppose qu'un système expert travaille sur des objets dont les catégories sont décrites en 2.3.1. La base d'objets contient un objet $\mathcal{B}$ de catégorie **Bureau** et n objets $\mathcal{M}_1 \ldots \mathcal{M}_n$ de catégorie **Meuble**. Le champ *elements_mobilier* de $\mathcal{B}$ peut contenir toute combinaison de l'ensemble $\mathcal{M}_1 \ldots \mathcal{M}_n$. A tout moment, au cours du raisonnement, des objets de catégorie **Meuble** peuvent être ajoutés ou retirés de la liste *elements_mobilier*($\mathcal{B}$). De plus, le champ *prix* de chaque objet de catégorie **Meuble** peut lui aussi être re-évalué par le système expert. Nous voulons à présent établir la contrainte suivante qui exprime de façon naturelle que le coût total de l'ameublement d'un bureau est la somme du prix de tous les meubles composant cet ameublement. Cette contrainte s'exprime de la façon suivante :

$$cout\_mobilier(\mathcal{B}) = \sum_{\mathcal{M}_i \in elements\_mobilier(\mathcal{B})} prix(\mathcal{M}_i) \tag{3.1}$$

On suppose enfin qu'au moment où on établit la contrainte, on a:

$$elements\_mobilier(\mathcal{B}) = (\mathcal{M}_1, \mathcal{M}_3, \mathcal{M}_7)$$

Que devons nous faire pour exprimer la contrainte 3.1 dans un langage de contraintes classique à base de dépendances ? Deux approches se présentent spontanément, qui vont se réveler fausses toutes les deux. La première consiste à créer une dépendance entre les champs *elements_mobilier* et *cout_mobilier* de l'objet $\mathcal{B}$. Cette dépendance permettra d'actualiser *cout_mobilier*($\mathcal{B}$) chaque fois que *elements_mobilier*($\mathcal{B}$) sera modifié. Par exemple, le fait d'ajouter le meuble $\mathcal{M}_2$ au mobilier de $\mathcal{B}$ va déclencher le calcul de la nouvelle valeur de son champ *cout_mobilier*. Mais que se passera-t-il si le prix de $\mathcal{M}_1$ est modifié au cours des inférences futures du système expert? Comme il n'existe pas de dépendances entre le champ *prix* de $\mathcal{M}_1$ et le champ *cout_mobilier* de $\mathcal{B}$, la contrainte ne sera pas réveillée et par conséquent, elle restera dans un état incohérent. Qu'à cela ne tienne. Nous allons prendre le problème sous un autre angle et créer une dépendance entre chaque champ *prix* des objets dans *elements_mobilier*($\mathcal{B}$) et le champ *cout_mobilier* de $\mathcal{B}$. Le problème est alors reporté sur le champ *elements_mobilier* de $\mathcal{B}$ car si celui-ci est modifié, le champ *cout_moblier* de $\mathcal{B}$ ne sera pas mis à jour. En désespoir de cause, on peut penser à ajouter le champ *elements_mobilier* de $\mathcal{B}$ à l'ensemble des dépendances de la contrainte. Le problème n'est pas résolu pour autant puisque l'ajout ou le retrait d'un meuble dans *elements_mobilier* ne s'accompagnera pas d'une mise à jour en conséquence de l'ensemble des dépendances.

Au-delà d'une simple combinaison, nous ressentons donc le besoin d'une interaction entre les deux types de dépendances que nous venons de présenter. Ainsi, afin de rester cohérent dans un environnement dynamique, des contraintes telles que 3.1 doivent être gérées explicitement par l'utilisateur. Cela veut dire que, quand un lien de dépendance doit être ajouté ou retiré, la contrainte qui possède ou nécessite ce lien devient obsolète. L'utilisateur doit alors la retirer et la remplacer par une nouvelle[2]. Nous prétendons que cette obligation n'est pas acceptable dans le cadre d'environnements aussi dynamiques que les bases d'objets des systèmes experts.

[2] Ce problème est une généralisation de celui évoqué dans les conclusions de la thèse de Steele ([Ste80] 10.2.1). Il propose l'implantation d'un système dans lequel les contraintes pourraient être créées sans que toutes ses dépendances soient connues.

En effet, au cours des premières phases du raisonnement, les objets de la base sont sujets à de fréquents changements au niveau de leurs sous-parties, reflétant ainsi la recherche d'une alternative de conception par exemple. Ceci ne doit en aucun cas être préjudiciable au maintien de la cohérence. L'utilisateur doit pouvoir exprimer des contraintes de manière abstraite et laisser au système de contraintes l'initiative de la gestion des dépendances. Pour satisfaire ceci, nous pouvons remarquer que le fait de dire que l'ensemble des dépendances de la contrainte est égale à l'ensemble des champs *prix* de tous les meubles de $elements\_mobilier(\mathcal{B})$ n'est rien d'autre qu'une contrainte. Cette contrainte agit au niveau *méta* puisqu'elle contraint une partie d'une autre contrainte.

3.2.2 Une solution avec méta-contraintes

Pour la clarté de l'exposé, nous introduisons ici quelques notations. On notera $\&(c, \mathcal{O})$ l'objet de catégorie `Attribut` qui représente le champ c de l'objet $\mathcal{O}$. On définit par ailleurs une fonction $\star$ de déréférençage des attributs telle que :

$$\star(\&(c, \mathcal{O})) = c(\mathcal{O})$$

Muni de ces conventions, on peut aisément exprimer la contrainte 3.1 par deux contraintes: une au niveau objet et l'autre au niveau méta. La contrainte de niveau objet est représentée par le relieur $\mathcal{C}_1$ et telle que :

$$cout\_mobilier(\mathcal{B}) = \sum_{\mathcal{D} \in dependances(\mathcal{C}_1)} \star(\mathcal{D})$$

Cette contrainte va assurer que le coût du mobilier est la somme de la valeur de ses dépendances. Ces dépendances sont les attributs représentant les champs *prix* des meubles de la liste $elements\_mobilier(\mathcal{B})$. Au moment où la contrainte est exprimée, ces dépendances sont $\&(prix, \mathcal{M}_1)$, $\&(prix, \mathcal{M}_3)$ et $\&(prix, \mathcal{M}_7)$. D'autre part et afin de prendre en compte l'évolution de cette dernière, on doit établir la méta-contrainte sur $\mathcal{C}_1$ qui sera représentée par le relieur $\mathcal{C}_2$, telle que :

$$dependances(\mathcal{C}_1) = \bigcup_{\mathcal{M}_i \in elements\_mobilier(\mathcal{B})} \&(prix, \mathcal{M}_i)$$

Ces deux contraintes sont exprimées une fois pour toutes et vont permettre de maintenir la contrainte 3.1 quels que soient les changements susceptibles d'intervenir sur le champ *elements_mobilier* de $\mathcal{B}$ ou sur le *prix* des $\mathcal{M}_i$.

3.2.3 Autres applications

On se rappelle le problème concernant la référence de variables par des chemins à travers des hiérarchies de sous-parties, évoqué en 2.2. Ce problème se ramène aussi à la gestion de l'ensemble des dépendances d'une contrainte et peut donc se résoudre à l'aide de méta-contraintes. Supposons par exemple que l'on veuille établir la contrainte

$$surface(emplacement(\mathcal{M}_6)) > 12$$

Nous allons exprimer cette contrainte à l'aide de deux relieurs. Le relieur $\mathcal{C}_3$ représente la contrainte qui assure que la valeur de sa dépendance est supérieure à 12 :

$$\star(dependances(\mathcal{C}_3)) > 12$$

Cette dépendance est la surface de l'emplacement de $\mathcal{M}_6$. Elle doit être modifiée selon l'emplacement du meuble. Cette modification est assurée par une seconde contrainte, représentée par un relieur $\mathcal{C}_4$, telle que :

$$dependances(\mathcal{C}_3) = \&(surface, emplacement(\mathcal{M}_6))$$

Bien sûr, ceci est généralisable pour un chemin de longueur n et on peut envisager de générer automatiquement la description des méta-contraintes. On obtient ainsi un système beaucoup plus uniforme que celui réalisé grâce à des attachements procéduraux par SOCLE [Har86].

Jusqu'ici, nous avons montré l'utilité de contraindre le champ *dependances* d'une contrainte. Nous étudions à présent quel peut être l'intérêt de contraindre ses autres champs.

Chaque objet possède un champ *existence* (c.f. 2.3.1) qui définit son existence dans la base d'objets courante. "$existence(\mathcal{O}) = ()$" indique que $\mathcal{O}$ n'est pas présent dans la base d'objets. Le fait de contraindre le champ *existence* d'un objet **Relieur** va permettre d'exprimer des *contraintes conditionnelles.* Une contrainte conditionnelle est une contrainte dont l'existence est subordonnée à la validité d'une condition. Elle doit donc cesser d'être active lorsque sa condition n'est plus vérifiée et être redéclenchée dès que cette condition redevient valide. Ainsi, supposons que deux tables $\mathcal{T}_1$ et $\mathcal{T}_2$ aient été liées par la contrainte **meme-hauteur** lors d'une application de la règle **uniformiser** décrite en 2.3.2. Soit $\mathcal{C}$ le **Relieur** représentant cette contrainte. Nous voulons que cette contrainte ne soit effectivement prise en compte que lorsque les deux tables appartiennent au même bureau. Pour cela, il nous suffit d'établir une méta-contrainte portant sur le champ *existence* de $\mathcal{C}$ telle que:

$$existence(\mathcal{C}) = eq(bureau(\mathcal{T}_1), bureau(\mathcal{T}_2))$$

Dès que $\mathcal{T}_1$ et $\mathcal{T}_2$ ne seront plus dans le même bureau, l'existence de $\mathcal{C}$ sera () et la contrainte cessera de peser sur les hauteurs des tables. Réciproquement, lorsque les deux tables seront remises dans le même bureau, $\mathcal{C}$ redeviendra active et les hauteurs des deux tables seront réajustées si nécessaire.

Bien qu'il soit aussi possible de piloter la valeur du champ *relation* d'une contrainte, nous ne voyons pas l'intérêt, dans des applications usuelles des contraintes, de modifier dynamiquement la relation existant entre un ensemble d'attributs donné.

3.2.4 Satisfaction avec méta-contraintes

Nous allons voir dans cette section que les contraintes réflexives, si elles apportent beaucoup de souplesse et de simplicité au niveau de l'expression, nécessitent un algorithme de satisfaction qui n'est pas aussi direct que l'algorithme classique.

N.B. : Les remarques que nous exprimons ici n'ont rien de formel et restent très intuitives.

Retour arrière lors de la construction d'une alternative

Comme nous l'avons exposé au chapitre précédent, le centre de notre système de satisfaction de contraintes est un algorithme basé sur le principe de propagation. Ce principe peut être vu comme un ordre particulier pour le parcours d'un graphe, à l'instar de la profondeur d'abord ou de la largeur d'abord [Per87]. L'ordonnancement est basé sur des informations purement locales extraites des noeuds du graphe. Que le parcours se fasse en une phase avec calcul en chaque noeud ou en deux phases par planification puis recalculs, il n'est jamais remis en cause par une exploration plus en avant.

Lorsque l'on s'attache à satisfaire des méta-contraintes, le parcours du graphe peut impliquer des modifications de sa topologie puisque les ensembles de dépendances ou l'existence même

des contraintes, et donc des noeuds du réseau, est calculée dynamiquement. La construction des graphes de propagation que nous avions décrite en 1.4.2 va à présent se faire par une stratégie d'essai-erreur assistée par retour arrière. En effet, on doit remettre en cause cette construction lorsque l'on modifie dynamiquement une partie de la structure du réseau de contraintes après avoir déjà parcouru cette même partie. Explicitons les occurences précises et les conséquences de ce retour arrière.

Au cours de la construction d'un graphe de propagation, l'algorithme procède pour ses propres besoins à la lecture de la valeur de certains des attributs des contraintes. Il faut en effet s'assurer qu'une contrainte existe avant de la traiter puis déterminer quelles sont ses dépendances pour continuer à propager. Ces actions impliquent respectivement la lecture du champ *existence* et *dependances* du relieur représentant la contrainte. On peut donc considérer que les attributs utilisés pour la construction d'un noeud sont des antécédents d'*expansion* de ce noeud, c'est à dire que leur valeur doit être connue avant que le noeud puisse être expansé[3]. Si l'antécédent d'expansion dont on requiert la valeur figure comme conséquent d'un noeud du graphe de propagation, sa valeur courante ne peut être utilisée immédiatement. Il faut d'abord lancer son évaluation afin de connaître sa nouvelle valeur. Cette évaluation entraine l'évaluation récursive de la fermeture transitive des antécédents du noeud considéré comme on l'a vu en 1.4.2. Le graphe peut donc comporter, avant sa complétion, des attributs calculés et dont la valeur a été utilisée.

Il peut alors arriver que l'algorithme détruise la correction causale du graphe de propagation soit en ajoutant un antécédent à un noeud déjà évalué soit en créant un noeud dont un des conséquent est un antécédent d'expansion d'un autre noeud du graphe. Cela correspond à la remise en cause de la valeur d'un antécédent d'expansion sur laquelle s'est basé une partie de la propagation. Ainsi et afin que la sémantique des dépendances entre les noeuds reste cohérente, il faut effectuer un retour arrière sur le graphe de propagation à partir du noeud qui a sollicité l'évaluation de l'antécédent d'expansion devenu obsolète. Ce retour arrière a pour rôle d'une part d'éliminer du graphe de propagation tous les sommets qui dépendent uniquement du noeud incriminé et d'autre part de replacer ce noeud dans l'ensemble des noeuds à expanser de l'alternative en cours de développement, ceci afin de recommencer sa propagation avec une valeur correcte de son antécédent d'expansion.

Causes de déclenchement d'une contrainte

Toujours dans les systèmes de propagation classiques, une contrainte n'est déclenchée qu'après qu'un attribut impliqué dans celle-ci est modifié. Avec notre représentation, une autre raison de réveiller une contrainte peut être la modification d'un de ses propres champs. En effet, si la valeur d'un champ d'une contrainte est modifiée au cours de la propagation, on doit considérer que c'est une nouvelle contrainte qui est installée dans la base d'objets et on doit donc déclencher son application. Un exemple naif de propagation peut illustrer ceci. Supposons qu'une règle d'inférence ajoute $\mathcal{M}_6$ à la liste $elements\_mobilier(\mathcal{B})$. La contrainte $\mathcal{C}_2$ impliquant $elements\_mobilier(\mathcal{B})$ est déclenchée la première. Elle est satisfaite en recalculant la valeur de $dependances(\mathcal{C}_1)$. La modification d'un champ de $\mathcal{C}_1$ déclenche celle-ci à son tour et va recalculer la nouvelle valeur de $cout\_mobilier(\mathcal{B})$.

[3] jusqu'à présent, les antécédents d'un noeud étaient des antécédents de *calcul* c'est à dire que leur valeur devait être connue avant de pouvoir calculer la valeur des conséquents du noeud.

3.3 Le système PROSE

Au sein du générateur SMECI, nous avons implanté notre système de satisfaction de contraintes sous la forme d'un système expert appellé PROSE[4]. Nous justifions ici ce choix et nous étudions la réalisation de ce système.

3.3.1 Réalisation

Comme son nom veut l'indiquer, le générateur SMECI permet l'écriture de systèmes **multi-experts**. Le terme multi-experts signifie ici que SMECI est capable de gérer le raisonnement de plusieurs systèmes experts indépendants et d'entrelacer leur processus en les faisant s'observer et éventuellement interagir. Ceci est initialement rendu possible par la réification[5] du raisonnement qu'effectue le moteur d'inférence (c.f. section 2.3.1). Plusieurs représentations de raisonnements peuvent ainsi cohabiter de façon simultanée. A leur tour, les systèmes experts SMECI sont des objets de catégorie SE (pour Systeme-Expert) qui développent leur propre raisonnement. Au sein de chaque système expert, le raisonnement passé, présent et futur est décrit respectivement par un *arbre d'états*, un *état courant* et un *agenda*. Outre ces descriptions, un objet de catégorie SE est caractérisé par une base de connaissance propre qui comporte la définition de la tâche à réaliser et la stratégie de raisonnement à utiliser.

La réification de la plupart des concepts de SMECI est bien sûr une porte ouverte sur l'utilisation de différentes formes de méta-raisonnement. En effet, le moteur de SMECI peut unifier sur des objets de catégorie SE et accéder à leur raisonnement en unifiant sur leurs états. Un système expert se_1 peut ainsi appliquer des règles qui vont manipuler l'état interne d'un autre système expert se_0. PROSE s'implante donc naturellement comme un système expert indépendant raisonnant sur un système observé donné. Notons que cette approche a déjà été utilisée en SMECI pour élaborer un système expert explicateur [DCH87] qui observe le système expert à expliquer.

Le raisonnement de PROSE sera lancé à l'issue de la création de chaque nouvel état du système observé et aura pour but, partant des modifications induites par l'état, de trouver le meilleur ensemble d'affectations de variables qui restaure la cohérence de la base d'objets du système observé.

3.3.2 Avantages

La motivation principale de l'écriture de PROSE est la réduction de la taille du code source ainsi que la facilité de maintenance de ce code. D'autre part, nous avons utilisé SMECI comme un langage de programmation pour la génération et le parcours d'arbres d'alternatives. Par exemple, le changement de stratégie pour la génération des alternatives de propagation se fait naturellement en changeant la stratégie du système expert. Ceci constitue une opération élémentaire en SMECI puisqu'il suffit de changer l'objet `Strategie` attaché à l'objet représentant le système expert. On profite aussi de la gestion d'états multiples réalisée par SMECI, c'est à dire que l'on n'a pas à se préoccuper de conserver des contextes pour les affectations effectuées dans une alternative puisque les affectations dans un état n'ont pas d'existence dans les états pères.

3.3.3 Implantation

Une application SMECI est programmée par une base de connaissances. Il en va de même pour le système PROSE qui définit :

[4] **PRO**pagation pour et par **S**ystème-**E**xpert

[5] Transformation d'une *idée* en une entité manipulable en tant qu'*objet* par un programme [Fer87].

- des catégories pour modéliser les noeuds du graphe de propagation ainsi que les alternatives.
- des règles qui effectuent les actions de la propagation en créant et en manipulant des noeuds et des alternatives.
- des méthodes définissant certains comportements pour les noeuds et les alternatives.
- des objets `Tache` et `Base_de_regles` qui définissent le contrôle

Le système propagateur actuel représente 10 règles réparties en 6 bases de règles et approximativement 150 lignes de code pour les méthodes. On voit donc que le code est très compact surtout si on le compare à celui de la version écrite en LE_LISP qui totalise environ 1200 lignes. Il y a bien sûr un prix à payer pour cette approche claire et concise : le temps d'exécution de la propagation se trouve légèrement augmenté.

3.4 Conclusion.

La représentation que nous avons donné de nos contraintes leur permet de s'adapter à des environnements tels que des bases d'objets complexes des systèmes experts, tout en conservant leur sémantique à travers l'évolution des liens entre objets. Cette représentation nous a amenés à considérer la notion de réfléxivité. Cette investigation doit encore être approfondie. On envisage en effet de découvrir le caractère réfléxif de l'algorithme de satisfaction. Pour cela, il nous faut définir cet algorithme comme un interprète à part entière dont les programmes interprétés seraient des liste de contraintes à déclencher. L'interprétation d'une méta-contrainte ferait alors passer l'interprète dans un état introspectif qui effectuerait le retour arrière sur le graphe (l'environnement de l'interprète) et qui modifierait la liste des contraintes à déclencher (c'est à dire la continuation). On envisage aussi d'appliquer les méta-contraintes à des activités de contrôle de l'algorithme de propagation comme cela est évoqué dans [Bor79].

A Propagation : algorithmes et exemple

A.1 Construction et évaluation des graphes de propagation

Nous décrivons ici les algorithmes évoqués en 1.4.2 pour la génération des graphes de propagation et l'évaluation d'un noeud du graphe. L'expression que nous en livrons n'est pas optimisée afin de rester lisible.

Structures Outre les variables et les contraintes du problème, l'algorithme utilise les trois structures suivantes:

- des *noeuds* comprenant les champs
  antecedents : liste de noeuds
  contrainte : contrainte
  consequents : liste de variables
  marque : booléen
- des *alternatives* comprenant les champs
  a_expanser : liste de noeuds
  expanses : liste de noeuds
- un *sac* comprenant le champ
  alternatives : liste d'alternatives

On dispose des fonctions *creer_noeud*, *creer_alternative* et *creer_sac* pour la création de ces structures. On dispose d'autre part d'une fonction *copier* pour les noeuds et les alternatives qui permet d'en dupliquer un exemplaire donné. Enfin, on peut déposer une alternative A dans le sac S par $deposer(A,S)$ et extraire une alternative du sac par $extraire(S)$.

Propagation Lors du lancement de la propagation, on regroupe dans Δ_C (resp. Δ_v) toutes les contraintes créées (resp. toutes les variables modifiées) depuis la dernière propagation. On exécute d'abord une procédure d'initialisation qui va définir et organiser l'ensemble initial des noeuds à expanser. Cette procédure construit un noeud racine qui symbolise l'action de l'utilisateur sur le réseau de contraintes. Ce noeud n'est associé à aucune contrainte et sa liste de conséquents est l'ensemble des variables modifiées c'est à dire Δ_v.

algorithme $Initialisation(S : sac, \Delta_C : liste\_de\_contraintes, \Delta_v : liste\_de\_variables)$
$A \leftarrow creer\_alternative()$,
pour-tout $C \in \Delta_C$ faire
    $N \leftarrow creer\_noeud()$,
    $contrainte(N) \leftarrow C$,
    $a\_expanser(A) \leftarrow a\_expanser(A) \cup \{N\}$,
si $\Delta_v \neq \emptyset$
alors $N \leftarrow creer\_noeud()$,
    $marque(N) \leftarrow$ t
    $consequents(N) \leftarrow \Delta_v$,
    $expansion(A, N)$,
$deposer(A, S)$.

Une fois le sac initialisé, on peut lancer le développement des alternatives. L'ensemble des groupes de variables modifiables pour la satisfaction d'une contrainte C sont donnés par $modifiables(C)$.

algorithme $Propagation(S : sac)$
tant-que $alternatives(S) \neq \emptyset$ faire
$A \leftarrow extraire(S)$,
si $a\_expanser(A) = \emptyset$
alors A est une alternative complète.
sinon $N \leftarrow premier(a\_expanser(A))$,
$\mathcal{E} \leftarrow \{ \varepsilon \in modifiables(contrainte(N)),$
$\forall v \in \bigcup_{aN \in antecedents(N)} consequents(aN), v \notin \varepsilon\}$,
si $\mathcal{E} \neq \emptyset$
alors pour-tout $\varepsilon \in \mathcal{E}$ faire
$A_\varepsilon \leftarrow copier(A)$,
$N_\varepsilon \leftarrow copier(N)$,
$consequents(N_\varepsilon) \leftarrow \varepsilon$,
$expansion(A_\varepsilon, N_\varepsilon)$,
$deposer(A_\varepsilon, S)$,
sinon $expanses(A) \leftarrow expanses(A) \cup \{N\}$,
$deposer(A, S)$.

Dans le cas où on trouve une alternative complète, on peut soit arrêter la recherche (on a trouvé une solution potentielle) soit continuer la génération des alternatives suivantes (afin de comparer plusieurs alternatives complètes par exemple). L'expansion d'un noeud N dans une alternative A est réalisée par :

algorithme $Expansion(A : alternative, N : noeud)$
$expanses(A) \leftarrow expanses(A) \cup \{N\}$,
pour-tout $v \in consequents(N)$ faire
pour-tout $C \in contraintes\_portant\_sur(v)$ faire
si $C \neq contrainte(N)$
alors si $\exists N_c \in a\_expanser(A) \cup expanses(A)$
tel que $contrainte(N_c) = C$
alors si $N \notin antecedents(N_c)$
alors $antecedents(N_c) \leftarrow antecedents(N_c) \cup \{N\}$
sinon $N_c \leftarrow creer\_noeud()$,
$contrainte(N_c) \leftarrow C$,
$antecedents(N_c) \leftarrow \{N\}$,
$a\_expanser(A) \leftarrow a\_expanser(A) \cup \{N_c\}$.

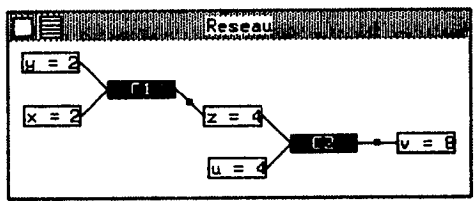
Evaluation L'algorithme de propagation ne boucle pas sur un réseau de contraintes comportant une circularité puisque chaque contrainte ne peut passer qu'une seule fois dans la liste des noeuds à expanser d'une alternative. Néanmoins, le graphe de propagation construit peut être circulaire. Il est donc nécessaire d'effectuer un test de circularité lors de l'évaluation des variables. Pour évaluer une variable, on cherche le noeud dont elle est un des conséquents sur lequel on applique l'algorithme suivant, avec une liste de noeuds déjà visités initialement nulle.

algorithme *Evaluation*(N : *noeud*, Vus : *liste_de_noeuds*)
si $\neg$*marque*(N)
alors si $N \in Vus$
alors le graphe de propagation comporte une circularité.
sinon pour-tout $N_a \in$ *antecedents*(N) faire
evaluation(N_a, $Vus \cup \{N\}$),
si *consequents*(N) = ()
alors tester *contrainte*(N)
sinon calculer *consequents*(N) avec *contrainte*(N),
marque(N) $\leftarrow$ t,

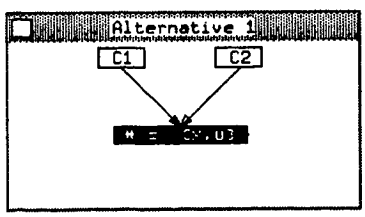
Lorsqu'on détecte une circularité lors de l'évaluation, on peut soit décider d'abandonner l'alternative soit d'appliquer une méthode numérique telle que la relaxation si la nature des contraintes s'y prête (c.f. 1.3.5). D'autre part, si le test d'une contrainte révèle une contradiction, l'alternative n'est pas une solution de la propagation.

A.2 Exemple de construction d'alternatives

Nous montrons ici un exemple du déroulement de la construction des alternatives de propagation. Pour ce faire, reprenons les deux contraintes $C_1 : z = x + y$ et $C_2 : v = z + u$ exprimées en 1.4. On rappelle le réseau qui les représente :



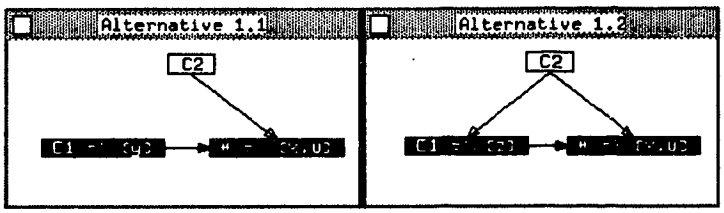
On suppose qu'on modifie initialement les variables x et u. On va donc créer un noeud racine dont les conséquents sont $\{x, u\}$ et dont l'expansion donne naissance à deux noeuds à expanser de contraintes C_1 et C_2. La pile d'alternatives contient alors l'alternative initiale 1 présentée ci-dessous[1].



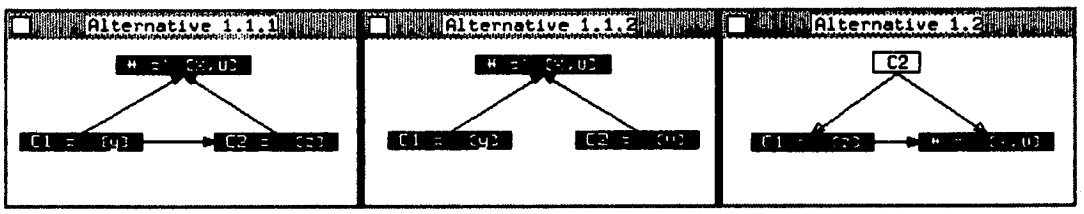
On dépile à présent l'alternative 1 et on expanse son premier noeud non-expansé. Celui-ci a pour contrainte associée C_1 et son ensemble de groupes de variables modifiables est $\{\{y\}, \{z\}\}$. On crée

[1] **Représentation des Alternatives :**
Les noeuds à expanser sont représentés en blanc et montrent le nom de la contrainte qui leur est associée. Les noeuds déjà expansés sont représentés en noir et montre le nom de leur contrainte ainsi que leurs conséquents c'est à dire les variables qu'ils recalculent. Les liens représentés figurent la relation d'antécédent entre les noeuds.

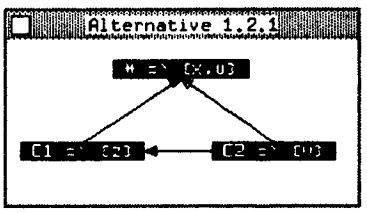
donc deux copies (alternatives 1.1 et 1.2) de l'alternative 1 auxquelles on associe respectivement $\{y\}$ et $\{z\}$ comme conséquents du noeud qu'on expanse. y n'intervient que dans C_1 et par conséquent, il n'y a ni lien, ni nouveau noeud à créer (C_1 n'est pas considérée puisque c'est la contrainte associée au noeud couramment expansé). En revanche, z intervient dans C_1 et C_2. Comme il existe déjà un noeud associé à C_2, on ajoute, en antécédent de ce noeud, le noeud expansé. Les deux copies sont successivement empilées et l'état de la pile est le suivant :



On procède ensuite de la même façon en dépilant l'alternative 1.1 et en considérant le noeud associé à C_2. Les groupes de variables modifiables sont ici $\{\{z\},\{v\}\}$. On donne ainsi naissance aux alternatives 1.1.1 et 1.1.2 et la pile devient :



1.1.1 est dépilée. C'est une alternative complète puisqu'elle ne comporte plus de noeuds à expanser. Il en va de même pour 1.1.2. On traite ensuite 1.2 en créant et en empilant 1.2.1 :



Enfin, 1.2.1 est dépilée et découverte complète. On trouve donc trois alternatives complètes (1.1.1, 1.1.2, 1.2.1) qui peuvent indifféremment être choisie pour restaurer la cohérence du réseau.

Bibliographie

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, 1974.

[Ba87] A. Borning and al. Constraint hierarchies. In *Proceedings of the OOPSLA 87*, 1987.

[Bat83] J. Batali. *Computational Introspection.* Technical Report 701, MIT AI Lab., 1983.

[Bor79] A. Borning. *ThingLab : A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, 1979.

[Cha87] J. Chailloux. *Le_Lisp Version 15.2.* 1987.

[CM84] W. Clocksin and C. Mellish. *Programming in Prolog, 2^{nd} ed.* Springer–Verlag, 1984.

[Cor88] O. Corby. *Un Tableau Réflexif pour la Coopération de Bases de Connaissances.* PhD thesis, Université de Nice, 1988.

[Cou78] *Cours Implémentation et Interprétation de Lisp.* Ecole de l'IRIA: Ecole de la Recherche, 1978.

[Dav87] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[DCH87] R. Dieng, O. Corby, and P. Haren. Un système expert explicateur. In *Proceedings of COGNITIVA 87*, 1987.

[deK86a] J. deKleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[deK86b] J. deKleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[Din86] M. Dincbas. Constraints, logic programming and deductive databases. In *Proceedings of the France-Japan AI Symposium 86*, 1986.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[EG87] S. Ervin and M. Gross. ROADLAB - a constraint based laboratory for road design. In *Proceedings of the Second International Conference on Applications of AI to Engineering*, 1987.

[Fer87] J. Ferber. Reflection in computational systems. In *Proceedings of COGNITIVA 87*, 1987.

[FL69] G. Friedman and C. Leondes. Constraint theory, part I, II & III : fundamentals. *IEEE Transactions on Systems Science and Cybernetics*, 5(2), 1969.

[Fre78] E. Freuder. Synthesizing constraint expression. *Communications of the ACM*, 21(11), 1978.

[GJV87] H. Gusgen, U. Junker, and A. Voss. Constraints in a hybrid knowledge representation system. In *Proceedings of the IJCAI 87*, 1987.

[Gos83] J. Gosling. *Algebraic Constraints.* PhD thesis, Carnegie–Mellon University, 1983.

[Gus86] H. Gusgen. *Foundation of a System for Constraint Satisfaction:* CONSAT. Technical Report, GMD Sankt Augustin, 1986.

[Gus88] H. Gusgen. Some fundamental properties of local constraint propagation. *Artificial Intelligence*, 36:237–247, 1988.

[Har86] D. Harris. A hybrid structured object and constraint representation language. In *Proceedings of the AAAI 86*, 1986.

[KM77] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP 77*, 1977.

[Lau86] J–L. Lauriere. *Intelligence Artificielle. Resolution de Problemes par, l'Homme et la Machine.* Editions Eyrolles, 1986.

[Mac77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mae86] P. Maes. *Reflection in an Object-Oriented Language.* Technical Report 86–8, Vrije Universiteit Brussel AI Lab., 1986.

[Mal87] J. Maleki. *ICONStraint : A Dependancy-Directed Constraint Maintenance System.* PhD thesis, Linkoping university, 1987.

[Per87] M. Perlin. On the computational equivalence of frame system and rule system. In *Proceedings of the US-Japan AI Symposium 87*, 1987.

[Ric83] E. Rich. *Artificial Intelligence, p. 176-184.* McGraw–Hill, 1983.

[Rit86] J–F. Rit. Propagating temporal constraints for scheduling. In *Proceedings of the AAAI 86*, 1986.

[SG87] D. Serrano and D. Gossard. Constraint management in conceptual design. In *Proceedings of the Second International Conference on Applications of AI to Engineering*, 1987.

[Sme88] *Smeci Version 1.3, Users' Reference Manual.* ILOG, 2 Av. Galliéni, F-94253 Gentilly, 1988.

[Smi82] B. Smith. *Reflection and Semantics in a Procedural Language.* Technical Report 272, MIT Laboratory for Computer Science., 1982.

[SS80] G. Sussman and G. Steele. CONSTRAINTS- a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[Ste80] G. Steele. *The Definition and Implementation of a Computer Programming Language Based on* CONSTRAINTS. PhD thesis, MIT, 1980.

[Ste87] S. Steel. On trying to do dependancy-directed backtracking by searching transformed state spaces (and failing). In *Proceedings of the AISB 87*, 1987.

[Sut63] I. Sutherland. *Sketchpad : A Man-Machine Graphical Communication System.* PhD thesis, MIT, 1963.

[Van87] K. VanMarcke. A parallel algorithm for consistency maintenance in knowledge representation. In B. DuBoulay, D. Hogg, and L. Steels, editors, *Advances in Artificial Intelligence - II*, North–Holland, 1987.

[Wal72] D. Waltz. *Generating semantic description from drawing of scenes with shadows.* Technical Report AI-271, MIT, 1972.

[ZMC87] R. Zabih, D. McAllester, and D. Chapman. Non-deterministic lisp with dependancy-directed backtracking. In *Proceedings of the AAAI 87*, 1987.

Table des matières

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 924

Programme 1

INTEGRATION D'OUTILS POUR L'EXPRESSION ET LA SATISFACTION DE CONTRAINTES DANS UN GENERATEUR DE SYSTEMES EXPERTS

Pierre BERLANDIER

Novembre 1988

RR-0924

Intégration d'outils pour l'expression et la satisfaction de contraintes dans un générateur de systèmes experts

Pierre Berlandier

INRIA Sophia Antipolis
06560 Valbonne

Travaux effectués dans le cadre du contrat n. 871448 passé par la Direction des Recherches et Etudes Techniques, Direction Scientifique - Section Soutien à la Recherche.

Intégration d'outils pour l'expression et la satisfaction de contraintes dans un générateur de système expert

Résumé

Les langages à base de contraintes apparaissent comme des outils efficaces pour la modélisation, la simulation et la résolution de problèmes. L'intégration d'un système de gestion de contraintes à base de dépendances dans un générateur de systèmes experts renforce la puissance de chacun des deux outils. Néanmoins, les langages de contraintes classiques se révèlent mal adaptés à la gestion d'environnements tels que les bases de faits des systèmes experts. Nous présentons dans ce papier une représentation des contraintes à caractère réflexif. Les méta-contraintes qui en résultent permettent la gestion explicite de l'ensemble des dépendances des contraintes de niveau objet. On conserve ainsi non seulement la cohérence des valeurs du réseau de contraintes mais aussi la cohérence du réseau lui-même vis-à-vis de la base de faits. Cette représentation implique des changements importants sur l'algorithme de propagation de contraintes classique.

Mots clés

Langages de contraintes, propagation de contraintes, systèmes experts, réflexivité.

Integrating Tools for Constraints Expression and Satisfaction in an Expert System Shell

Abstract

Constraint languages have proved to be efficient tools for modeling, simulating and solving problems. Integrating a dependency-based constraint language in an expert system shell results in power and flexibility benefits for both tools. Nevertheless, usual constraint languages turn out to be too weak to deal with such dynamic knowledge bases as expert systems ones. Therefore, we present in this paper a powerful constraint representation based on reflection. The resulting meta-level constraints allow explicit management of the dependency set of object-level constraints so that the dependency network is kept consistent according to the knowledge base evolvement. This representation induces substantial changes on the classic constraint propagation algorithm.

Keywords

Constraint languages, constraint propagation, expert systems, reflection.

Introduction.

" ... when a domain is well understood, it is often possible to describe the objects in the domain in a way that uncovers usefull, interacting constraints..."
Patrick H. Winston

Les outils

La résolution de nombreux problèmes est liée a l'exploration de graphes d'états. Or, dès que l'on sort du cadre d'exemples d'école, cette exploration est rapidement menacée par l'explosion combinatoire. Pour éviter de rencontrer ce monstre redouté, plusieurs types d'outils de l'intelligence artificielle contrôlent la recherche de solutions par des connaissances établies dans le domaine du problème à résoudre. Cette idée a principalement amené le développement des *systèmes experts* dont le raisonnement repose entièrement sur la donnée d'une quantité importante de connaissances.

Par ailleurs, la connaissance que l'on peut avoir sur un domaine s'énonce souvent sous la forme d'un ensemble de contraintes. Trouver une solution d'un problème consiste alors à isoler pour un ensemble de variables caractéristiques, les valeurs qui satisfont simultanément ces contraintes. De nombreux problèmes ont été réduits à des problèmes de satisfaction de contraintes. Parmi ceux-ci, on peut citer l'analyse de circuits [SS80], le filtrage dans l'analyse de scènes [Wal72] et certains aspects des processus de conception et de simulation [Sut63][Bor79]. Des outils adaptés à ce type de problèmes ont été écrits et ont donné naissance à de véritables *langages de contraintes* [Ste80] [Gus86].

Leur coopération

L'aspect déclaratif des règles d'inférence d'un système expert fragilise la cohérence de la base de faits au cours de la résolution d'un problème. En effet, à moins d'imposer un contrôle quasiment impératif sur l'application des règles, il est possible qu'une règle affecte inopinément le raisonnement antérieur du système expert. Certains démons des langages de schémas tels que les "*si-modifié*" permettent des contrôles élémentaires de cohérence. Néanmoins, leurs possibilités restent souvent assez réduites. Les systèmes experts montrent donc un besoin d'outils pour le maintien de la cohérence de leur base de faits au cours de leur raisonnement. C'est pourquoi nous avons établi une coopération entre un générateur de système expert et un langage de contraintes. Ce papier propose l'étude de la réalisation et les implications d'une telle coopération.

1 Satisfaction des contraintes

Au coeur de tout système de contraintes se trouve un mécanisme dont la tâche consiste à trouver un ensemble de valeurs telles que l'ensemble des contraintes exprimées soient satisfaites. Ce chapitre a pour but principal de présenter les détails d'un tel mécanisme: le mécanisme de propagation. Nous y explicitons aussi un algorithme de satisfaction que nous avons développé. Celui-ci reste basé sur le principe de propagation et de là, son mérite principal n'est pas l'originalité. Néanmoins, le manque d'adéquation des algorithmes existants avec les spécifications de notre problème justifie complètement son écriture ainsi que les choix qui le caractérisent.

1.1 Encore un problème NP-complet

Un problème NP-complet est un problème soluble par des algorithmes non-déterministes en temps polynomial c'est à dire des algorithmes qui parcourent parallèlement un nombre arbitraire de chemins qui, s'ils aboutissent, offrent une solution de complexité polynomiale. Pour cette classe de problèmes, tous les algorithmes déterministes connus fonctionnent en temps exponentiel. Ces problèmes, qui impliquent des recherches et des choix, relèvent donc typiquement des techniques de l'intelligence artificielle. [Ric83] va même plus loin en suggérant qu'on peut décrire l'intelligence artificielle comme étant la recherche de techniques permettant d'approcher une complexité polynomiale pour la résolution de problèmes NP-complets.

Le problème qui nous intéresse ici est lui-même NP-complet. On peut en trouver une démonstration dans [Lau86] ou [AHU74]. Ainsi, pour gagner en efficacité, l'algorithme de satisfaction devra perdre en généricité. Les systèmes de contraintes sont donc le plus souvent dédiés à un domaine précis afin de pouvoir intégrer le maximum d'heuristiques et réduire ainsi la complexité de la recherche.

1.2 Satisfaction par génération puis test

Cette section considère le problème de satisfaction sous sa forme la plus générale. Nous noterons $v_1, \ldots, v_n$ les variables du problème, $\mathcal{D}_1, \ldots, \mathcal{D}_n$ leurs domaines respectifs et $|\mathcal{E}|$ la cardinalité d'un ensemble $\mathcal{E}$. Il est important de souligner que les techniques décrites ici ne sont applicables que pour des variables ayant un domaine fini et discret. Elles supposent en effet qu'on peut représenter explicitement ou parcourir en un temps fini l'ensemble des k-uplets du produit cartésien des domaines de k variables quelconques du problème.

En toute généralité, un ensemble de contraintes peut s'exprimer de façon non-constructive[1], à l'aide d'un ensemble de prédicats unaires (de la forme $P_i(v_i)$) ou binaires (de la forme $P_{i,j}(v_i, v_j)$). Il s'agit alors de trouver des valeurs $x_1, \ldots, x_n$ telles que l'on ait:

$$P_1(x_1) \wedge \ldots \wedge P_n(x_n) \wedge P_{12}(x_1, x_2) \wedge P_{13}(x_1, x_3) \ldots \wedge P_{n-1,n}(x_{n-1}, x_n)$$

La stratégie la plus simple et la plus typiquement exponentielle consiste à passer en revue successivement les n-uplets du produit cartésien $\mathcal{D}_1 \times \ldots \times \mathcal{D}_n$ jusqu'à trouver un n-uplet satisfaisant. Il est clair que cette stratégie devient rapidement inutilisable puisqu'on peut tester jusqu'à $\prod_{i=1}^{n} |\mathcal{D}_i|$ combinaisons.

[1] Selon la terminologie de [Gus88], une contrainte *non-constructive* est une contrainte dont on peut seulement tester la validité. [Din86] parle de contrainte *passive*.

Retour arrière chronologique Au lieu de générer une valeur pour chaque variable puis de tester globalement l'ensemble des valeurs obtenues, il apparaît plus astucieux de combiner une instanciation et une évaluation progressive des contraintes du problème. L'ensemble des substitutions possibles pour chaque variable est géré par une pile. Dès que l'évaluation d'une contrainte révèle une contradiction, on remet en cause la valeur de la dernière variable instanciée. On va alors chercher la substitution suivante présente en sommet de pile, puis on reprend l'évaluation. Ce principe d'instanciation progressive avec retour arrière est utilisé par de nombreux systèmes de recherche non-déterministes. Il est notament au centre de la stratégie de recherche du langage PROLOG [CM84]. Le mécanisme est simple à implanter et permet de réduire considérablement l'espace de recherche surtout si la contradiction intervient assez tôt dans le processus d'instanciation. En effet, une contradiction intervenant après la détermination de la k-ième variable élimine l'examen de $\prod_{i \in I} |\mathcal{D}_i|$ combinaisons potentielles, où I est l'ensemble des indices des $(n-k)$ variables non encore instanciées.

Ordonnancement des contraintes Lorsqu'on doit alternativement évaluer des prédicats et instancier les variables intervenant dans ces prédicats, deux stratégies peuvent guider l'enchainement de ces actions :

1. *diriger les instanciations par l'évaluation.* Les prédicats sont alors traités dans un ordre donné et les variables sont instanciées au fur et à mesure qu'on en a besoin pour effectuer les évaluations. C'est le comportement adopté lors de l'évaluation d'une clause PROLOG.
2. *diriger l'évaluation par les instanciations.* Dans ce cas, c'est l'instanciation d'une variable qui va définir le prochain prédicat à évaluer, le but étant de ne pas entrelacer l'évaluation de prédicats dont les ensembles de variables sont disjoints.

Il est clair que dans le cadre du problème de satisfaction de contraintes, l'utilisation de la seconde stratégie permet d'améliorer les performances de la recherche (c.f. [Din86]). Il est par exemple plus judicieux d'évaluer la suite de prédicats

$$P_{i,j}(v_i, vj),$$
$$P_{j,k}(v_j, vk),$$
$$P_{l,m}(v_l, vm).$$

plutôt que

$$P_{i,j}(v_i, vj),$$
$$P_{l,m}(v_l, vm),$$
$$P_{j,k}(v_j, vk).$$

puisque, dans le second cas, une contradiction causée par la valeur de v_j dans $P_{j,k}$ va provoquer $|\mathcal{D}_l| * |\mathcal{D}_m|$ retours arrière avant de remettre en cause la valeur fautive.

Notons que l'ordonnancement de l'évaluation des prédicats se base sur une relation de dépendance implicite entre des ensembles de variables. Cette idée sera raffinée dans la section 1.3.4 où l'introduction de liens de dépendance, mais cette fois-ci au niveau des seules variables, débouchera sur une forme de retour arrière "intelligent".

Contrôle de cohérence des noeuds, des arcs et des chemins Le retour arrière chronologique reste malgré tout une méthode de recherche qui présente souvent un comportement grossier face au problème de satisfaction. Dans [Mac77], trois algorithmes sont proposés qui permettent d'appliquer un pré-traitement sur le domaine des variables et d'isoler de façon efficace certaines valeurs ou combinaisons de valeurs qu'on sait être incompatibles avec l'ensemble des contraintes. Cet ensemble est vu comme un graphe orienté dont les noeuds sont les variables et dont les arcs sont étiquetés par les prédicats. On détecte les incohérences dans ce graphe au niveau des

- **noeuds** On recherche les valeurs qui causent à elles-seules une contradiction au niveau d'un prédicat unaire c'est à dire les $x_i \in \mathcal{D}_i$ tels que $P_i(x_i)$ n'est pas vérifié. Celles-ci peuvent alors être immédiatement éliminées du domaine de la variable.

- **arcs** On suppose que les variables sont instanciées par ordre d'indice croissant. On recherche alors les valeurs $x_i \in \mathcal{D}_i$ telles que $\forall x_j \in \mathcal{D}_j$, $j > i$, $P_{i,j}(x_i, x_j)$ n'est pas vérifié. Ces valeurs constituent des cas pour lesquels le mécanisme de retour arrière va essayer toutes les combinaisons de $\mathcal{D}_{i+1} \times \ldots \times \mathcal{D}_j$ avant de s'apercevoir que x_i est une valeur impossible. On peut donc pareillement les éliminer.

- **chemins** On recherche les couples de valeurs $(x_i, x_j) \in \mathcal{D}_i \times \mathcal{D}_j$ tels que $P_i(x_i)$, $P_j(x_j)$, $P_{i,j}(x_i, x_j)$ sont vérifiés mais $\nexists x_k \in \mathcal{D}_k$ tel que $P_{i,k}(x_i, x_k)$, $P_k(x_k)$ et $P_{k,j}(x_k, x_j)$ soient simultanément vérifiés. Non seulement, comme pour le cas précédent, ces contradictions sont coûteuses à découvrir mais en plus elles peuvent être redécouvertes plusieurs fois. Il est donc important de prévenir la génération de tels couples de valeurs.

L'application de ces algorithmes permet d'éliminer rapidement de nombreuse causes de contradictions mal gérées par le retour arrière chronologique. Cependant, il doit être clair que l'existence de couples de valeurs cohérents au niveau des chemins n'implique pas l'existence d'une solution au problème de satisfaction. Dans [Fre78], les notions de cohérence au niveau des noeuds, des arcs et des chemins sont généralisées à celle de cohérence au niveau de sous-graphes d'ordre k du réseau de contraintes. Il est montré comment cette notion de k-cohérence, exploitée successivement pour k variant de 1 à n, permet d'isoler l'ensemble des n-uplets satisfaisant l'ensemble des contraintes. Ces techniques ont été utilisées avec succès pour des problèmes combinatoires tels le filtrage dans l'analyse de scènes [Wal72] ou la planification temporelle [Rit86].

1.3 Satisfaction par propagation

De nombreux types de contraintes peuvent être exprimés de façon constructive. Les relations associées à ces contraintes sont alors définies soit *extensionnellement* par énumeration de tous leurs k-uplets soit *intentionnellement* par la donnée d'un ensemble d'expressions fonctionnelles représentant chacune de leurs variables. Pour de telles contraintes, la donnée d'un certain nombre de variables peut entrainer la détermination des autres. Ainsi, on peut décrire la contrainte $a = b+c$ par les trois expressions :

$$a \leftarrow b + c$$
$$b \leftarrow a - c$$
$$c \leftarrow a - b$$

Connaissant par exemple les valeurs $a = 3$ et $c = 2$, on déduit naturellement $b = 1$. Cette valeur peut alors être *propagée* c'est à dire, utilisée par toutes les autres contraintes du problème impliquant la variable b. Cette remarque est à la base de l'algorithme de propagation.

1.3.1 Algorithme

Nous présentons dès maintenant l'algorithme dans sa version la plus épurée et dénué de toutes les subtilités qui ne seraient pas directement rattachées au principe de propagation locale.

On dispose d'un ensemble de variables dont les valeurs sont indéterminées et d'un ensemble de contraintes sur ces variables. L'algorithme utilise une pile de contraintes que nous appellerons *Contraintes-Declenchables*. Cette pile est initialement vide. Au fur et à mesure qu'on instancie

des variables du problème, on empile dans *Contraintes-Declenchables* toutes les contraintes attenantes à ces variables. Lorsqu'on décide d'essayer de satisfaire les contraintes, on exécute la boucle suivante:

Algorithme de propagation.

Tant que *Contraintes-Declenchables* n'est pas vide, dépiler le sommet de pile dans *Contrainte-Courante* puis appliquer une des règles suivantes:

1. Si toutes les variables de *Contrainte-Courante* sont déterminées et que la contrainte est déjà satisfaite, alors il n'y a rien à faire.
2. Si toutes les variables de *Contrainte-Courante* sont déterminées et que leurs valeurs contredisent la contrainte, alors il n'est plus nécessaire de continuer la propagation : les contraintes sont dans un état incohérent.
3. S'il n'y a pas assez de variables déterminées pour déclencher *Contrainte-Courante*, alors on ne peut rien faire. Lorsque de nouvelles variables participant à la contrainte seront déterminées, la contrainte reviendra en pile et sera alors reconsidérée.
4. Si *Contrainte-Courante* peut être déclenchée alors, déduire de nouvelles valeurs et empiler dans *Contraintes-Declenchables* toutes les contraintes attenantes aux variables déterminées, sauf bien sur *Contrainte-Courante* et celles qui sont déjà présentes dans la pile.

Si après l'exécution de cette boucle, certaines variables restent sans valeur, le système ne peut alors pas être résolu par seule propagation. Parmi les exemples suivants, nous présentons un tel cas de figure.

1.3.2 Exemples

L'exemple que nous donnons ici est simple et purement artificiel. Il a cependant l'avantage de pouvoir générer de nombreux cas de figure montrant l'éventail des possibilités et des déficiences de l'algorithme de propagation. Il implique quatre variables x, y, z, u et deux contraintes C_1 et C_2 telles que:

$$C_1 : x + y = z \quad C_2 : y + z = u$$

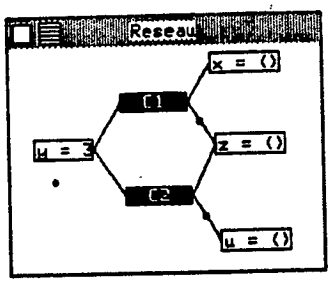
Notons que dans tout ce qui suit, un ensemble de contraintes est représenté par un graphe biparti dont les deux classes de noeuds sont les contraintes figurées en noir d'une part et les variables figurés en blanc d'autre part. Cette représentation, établie dans [FL69] offre une abstraction de la sémantique des contraintes qui permet de se concentrer sur leur aspect topologique et facilite la visualisation du flot de propagation.

1. On suppose que x et y sont connus. Cet exemple va nous permettre de constater que l'ordre dans lequel sont empilées, et donc traitées, les contraintes n'a pas d'importance quand

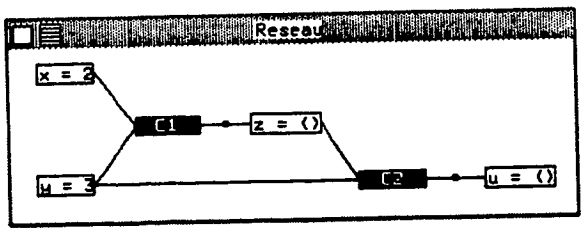
au résultat de la propagation. Avant le lancement de la propagation, la pile *Contraintes-Declenchables* peut en effet valoir $\{C_1, C_2\}$ ou $\{C_2, C_1\}$, selon qu'on a affecté x ou y en premier.

Si la pile vaut initialement $\{C_1, C_2\}$, C_1 est dépilée en premier dans *Contrainte-Courante*. Elle permet de déduire la valeur 5 ($x + y = 3 + 2$) pour la variable z. Les contraintes attenantes à z sont C_1 et C_2. La première est dans *Contrainte-Courante* et la seconde est déjà empilée. On n'empile donc aucune des deux. C_2 est alors dépilée et permet de déduire la valeur 8 pour u. De même que précédemment, C_2, attenante à u n'est pas empilée puisque c'est la contrainte courante. La pile est alors vide et la propagation s'arrête.

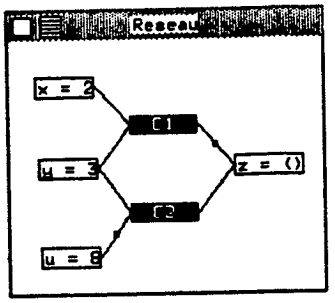
Dans le cas où *Contraintes-Declenchables* $= \{C_2, C_1\}$, C_2 est d'abord dépilée dans *Contrainte-Courante*. Seule la variable y est alors connue. On ne peut donc rien déduire. On passe à la contrainte C_1 qui permet elle de déduire la valeur de z et ré-empile C_2. On reconsidère alors C_2 qui permet à présent de déduire u puisque y et z sont maintenant connus.



2. On connait initialement les valeurs de x, y et u. *Contraintes-Declenchables* vaut au départ $\{C_2, C_1\}$. C_2 permet de déduire la valeur de z. Lorsque C_1 devient la contrainte courante, toutes ses valeurs sont alors connues et la satisfont. Il n'y a donc rien a faire et la propagation s'arrête.



3. Seule est connue la valeur de y. La détermination de cette valeur a empilé initialement dans *Contraintes-Declenchables* les contraintes C_1 et C_2. Il n'y a pas assez de valeurs connues pour déclencher l'une ou l'autre des contraintes. La propagation s'arrête alors sans rien déduire.



1.3.3 Attraits

L'algorithme de propagation offre des attraits largement reconnus puisqu'il est utilisé par une pléthore de systèmes sous une forme ou sous une autre. Parmi ces bons points, [Dav87] dégage les suivants, qui nous semblent particulièrement pertinents :

- L'algorithme est simple à coder et à étendre. Il est facile de comprendre et de suivre ses actions, de les expliquer à l'utilisateur et de les contrôler par un module extérieur intelligent.
- Il se comporte bien vis-à-vis des limitations de temps du calcul qu'on peut lui imposer. En effet, le fait d'interrompre le processus de propagation au cours de son déroulement permet de profiter des inférences qui ont déjà été accomplies.
- La nature locale du comportement des contraintes en fait un paradigme de programmation bien adapté aux traitements parallèles. Chaque contrainte peut être considérée comme un processeur indépendant qui ré-ajuste la valeur de ses arguments lorsqu'un d'entre eux est modifié. Cette approche rappelle les réseaux de processus communiquants présentés dans [KM77]. On trouve aussi dans [Van87] un exemple de parallélisation d'un algorithme de propagation pour un TMS[2].
- L'algorithme convient à des systèmes incrémentaux. En effet, entre deux applications du processus de propagation, on peut ajouter des contraintes dans le réseau. Leur effet est pris en compte dès la propagation suivante.

L'avantage dominant de l'algorithme de propagation reste qu'il permet de ne pas instancier *aveuglément* certaines des variables du problème. Néanmoins, afin de démarrer le processus, il est nécessaire que quelques unes d'entre elles soient arbitrairement valuées. Le choix de ces valeurs peut naturellement conduire à une contradiction après propagation. Il faut alors remettre en cause ces choix. Nous allons voir que ce retour arrière peut, lui aussi, ne pas s'effectuer aveuglément.

1.3.4 Retour arrière dirigé par les dépendances

D'une façon générale, lorsque le processus d'instanciation des variables du problème est dirigé par des inférences sur ces variables, on peut utiliser une forme de retour arrière efficace appelée retour arrière dirigé par les dépendances ou DDB[3] [SS80]. Sa mise en oeuvre implique la présence d'un mécanisme capable de mémoriser le déroulement des inférences accomplies par le

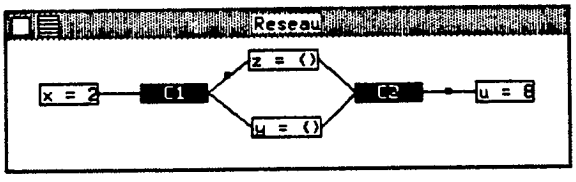
[2]Truth Maintenance System

[3]Dependency Directed Backtracking.

système. Ce système de gestion de dépendances ou RMS[4] construit un graphe de dépendances entre les inférences. Chaque noeud représente le résultat d'une inférence et est relié par un lien de dépendance aux noeuds qui ont permi et amené cette inférence. Ces derniers s'appèllent les *antécédents* du noeud. Le graphe est ensuite utilisé pour tracer les raisons d'une contradiction jusqu'à leur origine. Lorsqu'on rencontre une contradiction, les valeurs fautives qui l'ont impliquée sont retrouvées en remontant aux feuilles du graphe à travers les liens de dépendances (ce qui justifie la qualification "dirigé par les dépendances"). Cette recherche isole un ensemble de noeuds que nous appellerons *prémisses ultimes* du noeud contradictoire. Il faut alors choisir parmi les premisses ultimes le noeud dont on va modifier la valeur. Ce choix fait, on retire les instanciations qui dépendent du noeud choisi (toujours en utilisant le graphe de dépendances) et on relance le processus de recherche. Cette technique s'oppose au retour arrière chronologique qui, lors d'une contradiction, remet indifféremment en cause la dernière instanciation effectuée. Afin d'éviter de reproduire plusieurs fois les mêmes ensembles d'instanciations contradictoires, il faut enregistrer à chaque contradiction l'ensemble des valeurs des prémisses ultimes qui ont provoqué cette contradiction. Le DDB gère donc une liste couramment appelée *no-good-sets*, qui contient tous les k-uplets qui ne doivent pas être reproduits. Cette liste est ensuite consultée lors de chaque instanciation. Il faut remarquer que le fait de n'enregistrer que les k-uplets contradictoires permet d'utiliser le DDB pour des problèmes dont les variables ont des domaines infinis.

Le DDB est un mécanisme largement utilisé par divers systèmes de contraintes tels que ceux décrits dans [Ste80] ou [Mal87]. Il occupe une place centrale dans le fonctionnement des TMS tels que celui de Doyle [Doy79] ou l'ATMS de deKleer [deK86a,deK86b]. Enfin, d'autres utilisations du DDB sont décrites dans [ZMC87] et [Ste87].

1.3.5 Problèmes de circularités



La figure ci-dessus présente un cas où l'ensemble des variables connues est suffisant pour calculer les valeurs de y et z mais où l'algorithme de propagation est incapable d'accomplir une inférence. Ceci est dû à la présence d'une boucle dans le réseau, toutes les variables inconnues appartenant à cette boucle. Dans le cadre de contraintes d'ordre algébrique, il est possible d'offrir des solutions à ce problème de circularité.

Propagation symbolique Une première solution consiste à permettre aux contraintes de propager des expressions symboliques à travers le réseau [SS80]. Ainsi, supposons que l'on donne la valeur symbolique a à la variable y. La propagation de cette valeur à travers C_1 donne $z = a + 2$ puis $y = 8 - (a + 2)$ par C_2. On résout alors algébriquement l'équation $a = 8 - (a + 2)$ puis on substitue le résultat $a = 3$ dans les valeurs de y et z pour trouver $y = 3$ et $z = 5$.

Introduction de points de vue redondants Une autre solution au problème de circularité est de considérer le sous-réseau constitué par la boucle comme une seule et même contrainte et de compléter la définition de cette contrainte par une équation redondante. Dans notre exemple, on

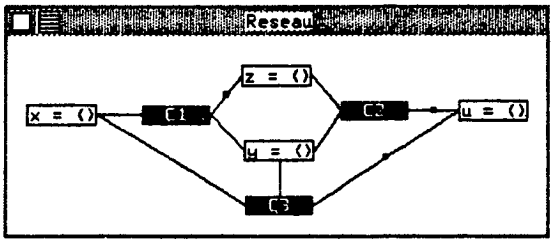
[4]Reason Maintenance System

va donc considérer la contrainte à quatre variables x, y, z et u telle que $x+y=z$ et $y+z=u$. Une simple combinaison de ces deux équations donne par exemple l'équation supplémentaire suivante:

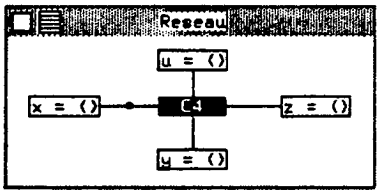
$$x+2*y=u \tag{1.1}$$

Grâce à ces trois équations et quelles que soient les combinaisons de deux variables données, les deux autres peuvent être déduites par propagation. Pour incorporer l'équation 1.1 au réseau, on peut soit :

- ajouter directement au réseau une contrainte correspondant à l'équation comme il est montré sur la figure ci-dessous avec la contrainte C_3. C'est la solution évoquée dans [Ste80] ou [Mal87].



- remplacer le sous-réseau circulaire par une contrainte à quatre variables synthétisant la sémantique des trois équations (c.f. contrainte C_4 sur la figure suivante). Cette technique est appliquée par le système de transformation algébrique de MAGRITTE [Gos83].



On peut remarquer que cette technique correspond en quelque sorte à une compilation dans le réseau de la technique précedente.

Relaxation En dernier ressort, on peut avoir recours à des techniques d'approximation numérique telles que la technique de relaxation. On choisit un ensemble de valeurs initiales pour les variables appartenant à la boucle. L'approximation se fait par itération d'ajustement de ces valeurs. Ces itérations sont contrôlées par une fonction d'erreur sur les valeurs ajustées. Selon le choix des valeurs initiales, ce processus peut diverger ou converger seulement très lentement. Cette lenteur fait que dans des systèmes comme SKETCHPAD [Sut63] ou THINGLAB [Bor79], la relaxation est utilisée si aucune autre solution n'a pu être appliquée.

1.4 Application incrémentale

Nous posons à présent le problème de gérer la cohérence d'un ensemble de variables à travers leur évolution. Cette évolution peut se traduire par l'ajout ou le retrait de variables ou de contraintes ou bien par le changement de la valeur de certaines variables. Ceci s'effectue de façon incrémentale, chaque étape du processus constituant ce que nous appelerons un *état*. L'ensemble des modifications qui sont apportées à un état Θ_i caractérise l'état Θ_{i+1} qui en résulte.

Jusqu'ici, l'algorithme de propagation faisait une partition claire entre les variables dont la valeur était connue et celles qui étaient inconnues. Cette partition nous permettait de reconnaitre quelles étaient les variables à calculer. A présent et si on veut garder un comportement déterministe, le problème va se poser de décider, pour chaque contrainte à satisfaire, laquelle (ou lesquelles) de ses variable *ajuster* en fonction des autres. Ce choix se révèle souvent difficile. Prenons pour exemple simple la contrainte C_1 présentée en 1.3.2 et supposons que dans un état Θ_i on ait $\{x = 2, y = 2, z = 4\}$. Dans l'état Θ_{i+1} suivant, on décide d'affecter $x \leftarrow 3$. Faut-il alors

- modifier y par $y \leftarrow 1$?
- modifier z par $z \leftarrow 5$?
- réinstaller $x = 2$?

Cette décision peut être aidée, de façon locale à chaque contrainte, par différents facteurs. Parmi ceux-ci, on distingue :

la sémantique de la contrainte L'exemple classique d'un choix de modification dirigé par la sémantique est celui du *point médiant* (c.f. [Bor79]) : trois points p_1, p_2 et p_m sont contraint d'être tels que p_m soit le milieu du segment $[p_1, p_2]$. Si on modifie maintenant la position de p_1 (figure 1.1), on a le choix entre bouger le point p_2 (figure 1.3) ou le point p_m (figure 1.2). Il est clair que la seconde solution est plus significative que la première. L'expression de la contrainte sous forme d'une équation lui a donc donné un caractère acausal alors que l'énoncé "p_m *est milieu du segment* $[p_1, p_2]$" distingait clairement un lien de causalité prédominant. Il est donc essentiel qu'outre la sémantique mathématique de la relation qu'elle représente, l'expression d'une contrainte puisse capturer l'existence de ce lien.

la date de modification Il est possible d'étiqueter chaque variable avec sa date de dernière modification. On peut alors choisir d'ajuster la variable la plus anciennement modifiée. On actualise ainsi progressivement l'ensemble des variables. On peut choisir au contraire d'ajuster la variable la plus récemment modifiée, entérinant ainsi la formation d'un lien fonctionnel entre cette variable et les autres partenaires de la contrainte.

l'arbitrage de l'utilisateur Si l'algorithme de satisfaction travaille en mode interactif, on peut faire intervenir l'utilisateur dans le choix de la variable. Néanmoins ce recours doit être très ponctuel et réservé à des cas jugés critiques. Le système risque sinon de devenir rapidement insupportable !

La section suivante s'intéresse à la résolution du problème de satisfaction incrémentale à travers différents systèmes existants. Nous allons voir comment certains de ces systèmes évitent la considération de choix locaux par l'utilisation de l'historique des inférences passées ou la définition d'une heuristique globale.

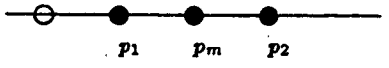


Figure 1.1: On déplace p_1 vers la gauche.

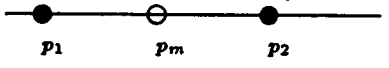


Figure 1.2: Le point milieu p_m s'ajuste pour satisfaire la contrainte.

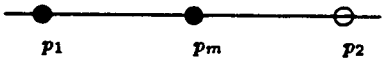


Figure 1.3: Le point p_2 s'ajuste pour satisfaire la contrainte.

1.4.1 Description de systèmes existants

THINGLAB

THINGLAB [Bor79] est un système écrit en SmallTalk. Il offre à l'utilisateur un environnement orienté-objet pour élaborer des simulations. Il permet pour cela de définir, de connecter, de visualiser et de faire évoluer des objets sous contraintes. L'utilisateur de THINGLAB peut attraper à l'aide de sa souris le point p_1 de l'exemple précédent, le déplacer et voir simultanément s'ajuster le point milieu au fur et à mesure du déplacement. Afin de rendre agréables ces interactions, il est essentiel que les techniques de satisfaction employées soient très performantes.

Expression des contraintes Le système permet d'exprimer, de façon statique lors de la définition d'une contrainte, un ordre de préférence quant à la variable à modifier en cas de déclenchement de cette contrainte. L'ordonnancement se fait en accord avec la sémantique de la contrainte. Ainsi, lorsque le processus de satisfaction doit faire un choix, il se réfère à cet ordonnancement pour prendre une décision. THINGLAB permet aussi de contrôler la directionnalité des contraintes exprimées. Pour cela, il faut définir (localement à la contrainte) certaines variables comme étant des *références*, c'est à dire des variables intervenant dans la contrainte mais qui ne pourront pas être modifiées pour la satisfaire.

Satisfaction des contraintes Motivé par un besoin d'efficacité, l'algorithme de satisfaction de THINGLAB présente l'originalité de s'exécuter en deux phases distinctes. La première planifie (sans les effectuer) les ajustements impliqués par les modifications de l'utilisateur et compile une méthode pour effectuer ces ajustements. La seconde phase n'est alors que l'invocation répétée de cette méthode. Par exemple, dès que l'utilisateur déplace un objet, la méthode réalisant la satisfaction des contraintes est générée et compilée à la volée une fois pour toutes. Elle est ensuite utilisée durant tout le déplacement (et pour des déplacements ultérieurs de même ordre).

La phase de planification d'une méthode de satisfaction s'exécute en faisant référence non pas aux valeurs des variables mais uniquement à la topologie du réseau de contraintes. Ceci peut engendrer du travail inutile lorsque, pour une des contraintes visitées, la modification imposée à une de ses variables ne provoque pas de violation de cette contrainte et donc n'implique pas l'ajustement d'une autre variable. Dans un système travaillant de façon incrémentale, on peut tester que la contrainte n'est pas satisfaite avant de poursuivre la propagation et ainsi parcourir un ensemble de noeuds minimal X pour assurer la satisfaction des contraintes. En THINGLAB, la phase de planification ne s'arrête que sur les feuilles du réseau de contraintes. On parcourt alors un ensemble de noeuds $X' \supset X$. Prenons pour exemple les deux contraintes :

$$C_1 : y = Ent(x) \quad C_2 : z = y^2$$

où $Ent(x)$ représente la partie entière de x. Initialement, on a $x = \frac{1}{3}$, $y = 0$ et $z = 0$. Modifions la valeur de x par $x \leftarrow \frac{1}{4}$. Si on propage en testant incrémentalement les contraintes, on s'aperçoit que la modification de x n'implique pas de changer la valeur de y. En revanche, si on procède à une planification sans calcul, on suppose que la modification de x entraine celle de y qui, à son tour, entraine celle de z.

Néanmoins, le surcroît d'opérations engendré par une planification se révèle moins grave qu'il n'y parait puisqu'aucun calcul coûteux n'est effectué sur les noeuds parcourus (on ne recalcule pas de valeurs lors de la planification). Des arguments plus subjectifs sont que des contraintes du type de C_1 sont assez rarement exprimées et qu'il est peu fréquent que les réseaux de contraintes soient d'une profondeur telle que $|X'| \gg |X|$.

CONSTRAINT

Le système décrit dans [Ste80] est assisté d'un RMS (c.f. 1.3.4). Il n'utilise que l'algorithme de propagation étudié en 1.3.1, c'est à dire que seules les variables non-valuées sont calculées. Lorsqu'à partir d'un état stable du réseau on décide de modifier la valeur d'une variable, il faut donc d'abord retirer sa valeur courante. Ce retrait s'effectue à l'aide du graphe de dépendances associé au réseau. On isole d'abord les prémisses ultimes de la variable à modifier. Parmi ces prémisses, on en choisit une. Ce choix se fait soit :

- en prenant la première rencontrée au cours de leur recherche. On peut considérer que c'est la prémisse la plus *directement* responsable de la valeur de la variable à modifier.
- en prenant celle qui implique le moins de répercussions sur les valeurs du réseau. On minimisera ainsi l'ampleur des modifications dans le réseau. Néanmoins, la détermination de cette prémisse peut se révéler couteuse puisqu'elle implique de déterminer l'ensemble de répercussion de chaque prémisse trouvée.
- indépendamment de la structure du réseau, en en choisissant une au hasard ou en demandant l'avis de l'utilisateur.

Une fois la prémisse choisie, on retracte la valeur des variables appartenant à la fermeture transitive de la prémisse selon le graphe de dépendances. On peut ensuite affecter la nouvelle valeur de la variable et lancer le processus de propagation. Nous allons fixer les idées à partir d'un exemple. Soient les deux contraintes

$$C_1 : z = x + y \quad C_2 : v = z + u$$

Le réseau initial est celui de la figure 1.4. Toutes les contraintes sont satisfaites. Il est donc stable. Le graphe de dépendance associé qui traduit l'histoire des inférences effectuées est donné par la figure 1.5. On décide de modifier la valeur de v. Pour cela, on recherche l'ensemble des prémisses ultimes de v qui, selon le graphe de dépendances, est $\{x, y, u\}$. Parmi ces prémisses, choisissons par exemple x. On retracte alors la valeur des répercussions de x. Ces repercussions apparaissent en grisé sur la figure 1.6. L'affectation de la nouvelle valeur de v suivie de l'application de la propagation va déduire les nouvelles valeurs de z et x. Il en résulte le nouveau graphe de dépendances montré sur la figure 1.7 où v est maintenant une des prémisses ultimes de x.

Le mécanisme de rétraction que nous venons de présenter n'implique qu'un seul choix par propagation : celui de la prémisse à incriminer. Chacune des prémisses ultimes d'une variable représente une séquence de choix possible sur les variables à modifier. L'ensemble de ces séquences est extrait à partir du graphe de dépendances. Il est donc imposé par la propagation précédente et constitue un sous-ensemble des séquences possibles. Par ailleurs, remarquons que tout comme le système précédent, le fait de planifier l'ensemble des modifications *avant* de les exécuter peut susciter plus de changements que nécessaires.

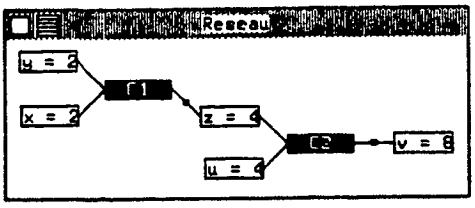


Figure 1.4: Réseau initial.

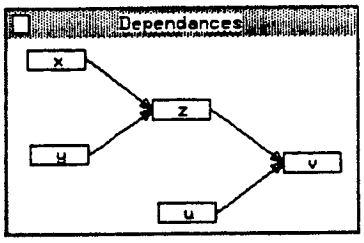


Figure 1.5: Graphe de dépendances initial.

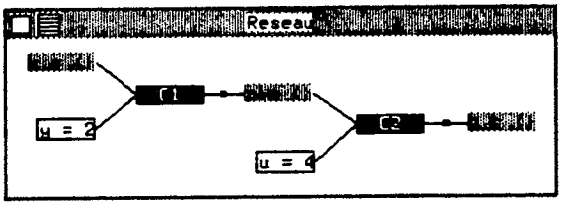


Figure 1.6: Valeurs retirées (en grisé).

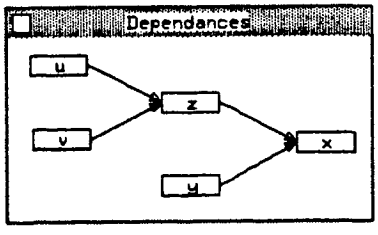


Figure 1.7: Graphe de dépendance aprés propagation.

MAGRITTE

MAGRITTE [Gos83] est un système permettant de définir des figures à l'aide de segments sur lesquels on exprime un ensemble de contraintes. Le système assure le maintien de la cohérence de la figure à la suite de chaque modification qu'elle peut subir. MAGRITTE contourne le problème du choix des variables à ajuster en appliquant la propagation de façon non-déterministe. L'algorithme de satisfaction est alors dirigé par un critère *global* qui est la recherche d'une bonne solution.

Qu'est-ce qu'une "bonne solution" ? c'est l'ensemble *minimum* de modifications qui permet de satisfaire l'ensemble des contraintes. Pour trouver cette solution, l'algorithme effectue une recherche en *largeur d'abord* sur toutes les alternatives de séquences de déclenchement de contraintes qui peuvent être progressivement générées en partant des modifications initiales. Il construit ainsi toutes les séquences de longueur 1, puis toutes celles de longueur 2, etc... jusqu'à trouver la première séquence complète qui satisfasse l'ensemble des contraintes. Ce sera là la solution optimale. Reprenons par exemple le réseau initial de la section précédente (figure 1.4). On décide de changer la valeur de x par $x \leftarrow 3$. La propagation de cette affectation peut engendrer successivement les quatre séquences illustrées par les figures 1.8, 1.9, 1.10 et 1.11 (les variables modifiées apparaissent en grisé). Dans le plus mauvais des cas, MAGRITTE va construire les deux premières séquences et choisir la seconde comme solution puisqu'elle permet de rétablir la cohérence du réseau par une seule modification.

L'inconvénient majeur de cette recherche est qu'elle peut impliquer beaucoup de gaspillage en temps de calcul. La naissance d'une séquence développe, en effet, un calcul propre qui peut être considéré comme un calcul perdu si cette séquence ne se révèle pas être un sous-ensemble de la séquence solution. Ainsi, dans notre exemple, le calcul de z pour la séquence de la figure 1.8 va s'avérer superflu puisque z ne fait pas partie des modifications prescrites par la solution choisie. Ceci n'a pas beaucoup d'importance si on travaille sur des contraintes simples comme des additions ou des multiplications ou si la profondeur du réseau est limitée. En revanche, la perte peut être importante si le rétablissement des contraintes requiert des calculs plus complexes.

Synthèse

En conclusion, il faut retenir :

- Comment la propagation incrémentale *choisit* les variables à ajuster.
  - **Par une stratégie locale** : en chaque contrainte visitée, on choisit quelle sera la prochaine variable modifiée (cas de THINGLAB).
  - **Par une stratégie globale** : on choisit une séquence complète de modifications parmi un ensemble de séquences donné. (cas de CONSTRAINT et de MAGRITTE).
- Comment la propagation incrémentale *effectue* les ajustements.
  - **En une passe** : les ajustements de valeurs sont alors effectués durant le parcours du réseau de contraintes. Cette solution oblige à gérer des environnements si on veut explorer plusieurs solutions (cas de MAGRITTE).
  - **En deux passes** : on planifie d'abord l'ensemble des ajustements et on les effectue ensuite. Il peut alors arriver que la phase de planification explore plus loin que nécessaire (cas de THINGLAB et de CONSTRAINT).

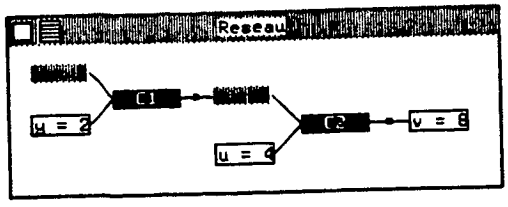


Figure 1.8: Séquence incomplète de longueur 1

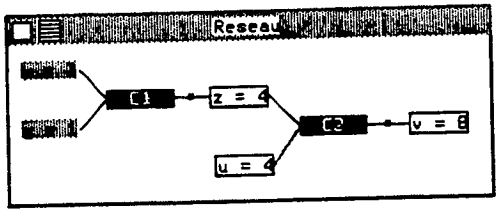


Figure 1.9: Séquence complète de longueur 1

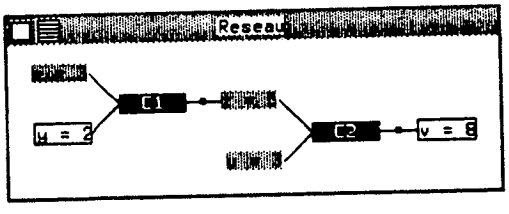


Figure 1.10: séquence complète de longueur 2

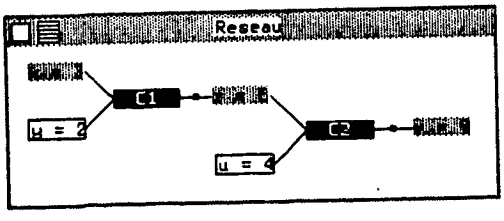


Figure 1.11: Séquence complète de longueur 2

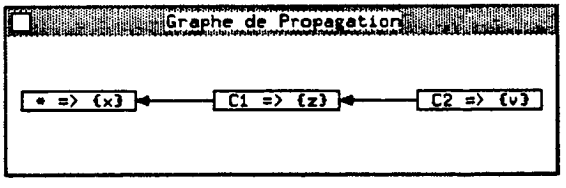
1.4.2 Une propagation plus souple

Le rétablissement local de la cohérence d'une contrainte se fait souvent en ajustant une seule des variables de la contrainte. Nous voulons élargir ce principe en offrant la possibilité de modifier plusieurs valeurs par contrainte déclenchée. [Ba87] montre l'utilité de cette possibilité sur l'exemple de la division entière où la donnée du diviseur et du dividende détermine le quotient *et* le reste. D'autre part, face à une modification, la plupart des algorithmes de propagation cherchent à recalculer immédiatement toutes les valeurs qui doivent être réajustées. Dans certains cas, on peut avoir besoin de ne recalculer qu'un certain nombre de variables qu'on juge intéressantes. [Mal87] donne l'exemple d'une fenêtre visualisant une partie d'un réseau d'objets sous contraintes. Lorsqu'on effectue une modification sur ce réseau, seule nous importe, dans un premier temps, la valeur des variables qui figurent dans la fenêtre. Ceci implique l'existence d'un mécanisme d'évaluation paresseuse des variables ajustées par propagation. Ce mode paresseux devra être utilisé avec précaution car les contradictions apportées par un ensemble de valeurs ne seront plus détectées dès leur introduction [EG87]. Nous décrivons maintenant un algorithme capable d'intégrer les deux caractéristiques précédentes.

Aperçu du mécanisme de propagation

La propagation paresseuse implique une décomposition en deux phases de l'algorithme. Comme en THINGLAB, la première phase va planifier les ajustements en construisant un graphe de propagation. Ce graphe définit les variables à recalculer ainsi que l'ordre de précédence des calculs. La deuxieme phase constitue l'exploitation de ce graphe. On pourra ainsi obtenir la valeur d'une variable modifiée en parcourant le graphe en chaînage arrière à partir de la variable demandée. Notons que, comme pour les méthodes compilées de THINGLAB, l'invocation répétée de la propagation de modifications portant sur un même ensemble de variables peut réutiliser le même graphe de propagation. Il n'y à alors plus qu'à effectuer les calculs.

Les graphes sont constitués de noeuds représentant le recalcul d'un ensemble de variables permettant de satisfaire une certaine contrainte. Un noeud est caractérisé par les attributs *antécedents*, *contrainte*, *conséquents* et *marque*. Ces attributs représentent respectivement l'ensemble des noeuds dont l'évaluation doit précéder celle du noeud lui-même, la contrainte qui effectue les ajustements, l'ensemble des variables à modifier[5] et enfin l'indicateur précisant si le noeud est déjà évalué ou non. Par exemple, la propagation de la modification de x dans le réseau présenté sur la figure 1.4 peut engendrer le graphe (élémentaire) suivant :



Ce graphe signifie que pour restaurer la cohérence du réseau, il faut modifier la valeur de z par C_1 et celle de v par C_2 et que la modification de v doit être postérieure à celle de z.

Un graphe en cours de construction est représenté par une *alternative*. Elle regroupe l'ensemble des noeuds d'un graphe partitionnés en deux catégories : d'un côté, les noeuds à expanser, c'est à dire les noeuds auxquels on doit associer un groupe de variables à modifier et de l'autre les noeuds déjà expansés. Lors de la propagation, les différentes alternatives produites sont rangées

[5] Cet ensemble peut être vide. Cela signifie alors que l'on doit simplement tester la validité de la contrainte associée au noeud.

dans un *sac*. La structure de ce sac est laissée à l'initiative de l'utilisateur. Dans l'implantation courante, ce sac est une *pile*. On réalise par conséquent une construction en profondeur d'abord des alternatives. Il est néanmoins très facile de modifier le module implantant la gestion du sac pour obtenir une construction de type largeur d'abord (en gérant le sac comme une *file*) ou meilleur d'abord (en réordonnant dynamiquement le contenu du sac au fur et à mesure des entrées).

On peut trouver en annexe une description précise de l'algorithme de génération et d'évaluation des alternatives ainsi qu'un exemple déroulant cette génération.

2 Contraintes et système expert

Nous rentrons avec ce chapitre dans le vif du sujet de ce rapport. Nous y traitons en détail de l'intégration d'outils pour l'expression, la création et le maintien de la cohérence de contraintes au sein d'un générateur de systèmes experts. Nous présentons tout d'abord les motivations d'une telle intégration puis nous décrivons précisémment comment nous l'avons implanté. Cette description intervient dans le cadre d'un générateur de systèmes experts particulier. Nous nous efforcerons néanmoins de ne faire appel qu'à des concepts de haut niveau afin de rendre notre approche assez générale pour s'adapter à d'autres générateurs.

2.1 Motivations

La plupart des générateurs de systèmes experts offrent la possibilité de représenter la connaissance de façon mixte. D'une part, l'expressivité des langages à base de schémas permet une représentation aisée de la connaissance structurale sur un domaine. D'autre part, les connaissances de type dynamique qui ne font pas partie de la connaissance sur un schéma particulier sont modélisées par des règles d'inférence. Les systèmes de contraintes offerts dans ces générateurs concernent habituellement le premier type de connaissance, c'est à dire la connaissance structurelle. L'utilisateur peut soit restreindre statiquement le domaine de validité de la valeur de certains attributs ou exprimer des relations entre les attributs d'une même instance de schéma qui seront testées et remises à jour à l'aide d'un mécanisme de démons. Néanmoins, ces mécanismes ne permettent pas d'établir ou de retirer incrémentalement des contraintes entre objets comme cela s'avère souvent nécessaire au cours d'un processus de raisonnement, surtout dans le domaine de la conception. En effet, une grande partie du processus de conception s'attache à la reconnaissance, l'expression et la satisfaction de contraintes [SG87] [EG87]. Les contraintes et leurs interactions sont alors trop nombreuses pour être gérées de façon explicite. Cette tâche peut incomber à un système de contraintes. Nous nous proposons donc d'intégrer un tel système dans un générateur de systèmes experts et d'obtenir une coopération entre les deux parties. De cette coopération, nous attendons principalement des bénéfices au niveau de la facilité et des possibilités d'expression des connaissances, du raisonnement et du maintien de la cohérence de la base de faits du système expert.

2.2 Travaux existants

Les travaux tendant à réunir les fonctionnalités d'un système à base de contraintes avec un autre type de système à base de connaissances semblent être assez rares. Nous en présentons ici deux exemples. [Gus86] présente l'intégration d'un système à base de contraintes nommé CONSAT [GJV87] dans un système de représentation de connaissances hybride. Ce dernier permet d'exprimer des connaissances sous forme de schémas, de règles d'inférence et de clauses PROLOG. La base de faits du système réunit donc des instances de schémas, des objets déduits par les règles et des faits assertés par les clauses PROLOG. Chaque élément communique avec le système de contraintes par l'intermédiaire d'un interprète propre à son formalisme de représentation. Quand un élément est modifié, CONSAT, qui utilise un algorithme de propagation, va réunir les valeurs des éléments appartenant au même réseau que celui-ci. La propagation de la modification va déduire un ensemble d'ajustements qui sera retransmis aux éléments de la base de faits toujours via leurs interprètes respectifs. Chaque formalisme est ainsi responsable de l'interprétation qu'il donne aux valeurs qui lui sont transmises. Par exemple, dans le cas des schémas, si la propagation retourne non

pas une valeur unique mais un ensemble de valeurs, cet ensemble peut être affecté à la facette *valeurs_possibles* de l'attribut modifié. Un des principaux attraits de cette intégration est que l'utilisation d'un système de contraintes évite une grande part de la programmation *explicite* qui serait nécessaire pour maintenir la cohérence de la base de faits.

Un autre exemple d'hybridation est donné dans [Har86]. Le système SOCLE qui y est présenté réunit le système de schémas FRL et le système de contraintes CONSTRAINT décrit en 1.4.1. Il est montré que les contraintes apportent un gain d'expressivité au système à base de schémas en permettant la définition de formules entre des attributs d'instances indépendantes. Les deux composants du système hybride communiquent ici par l'intermédiaire de cellules attachées aux attributs des schémas. Les contraintes sont installées entre ces cellules. Dès qu'une valeur est déterminée par une inférence du système de schéma, elle est répercutée sur la cellule correspondante. Réciproquement, lorsqu'une contrainte détermine la valeur d'une cellule, celle-ci est transmise à son attribut. Lors de la définition des contraintes, les variables sont désignées par des *chemins* à travers les hiérarchies de sous-parties des instances de schémas. On désignera par exemple "la `frequence_de_balayage` du `radar` du `systeme_de_controle`". Un problème survient quand on modifie la valeur d'un des attribut intervenant au milieu d'un tel chemin. Supposons en effet que l'on change le radar du système de contrôle. Il faut alors modifier le réseau en transférant la contrainte de la cellule correspondant à la fréquence de l'ancien radar à la cellule correspondant à la fréquence du nouveau. SOCLE résout le problème en plaçant, lors de la création des contraintes, des attachements procéduraux tout au long du chemin référençant la variable. Ces attachements sont chargés d'effectuer automatiquement le transfert de contrainte lorsque cela est nécessaire. Ce problème doit être considéré chaque fois que les contraintes sont effectivement attachées à une variable et que la variable à été déterminée par une référence susceptible d'évoluer.

2.3 Implantation

L'implantation de notre système a pour hôte le générateur de systèmes experts SMECI[1] [Sme88], développé à l'INRIA et écrit en Le_Lisp [Cha87]. Plutôt qu'une présentation détaillée de ce générateur, la section suivante constitue un dictionnaire qui décrit uniquement l'ensemble des concepts, des fonctionnalités et des mécanismes de SMECI utilisés par le système de contraintes. D'autre part, cette définition de la terminologie employée permettra de faire des rapprochements sans équivoques avec d'autres systèmes.

2.3.1 SMECI

SMECI est un outil qui permet de construire des systèmes experts à représentation mixte de la connaissance: schémas et règles. Son moteur d'inférences construit un arbre d'états qui est exploré à l'aide d'une stratégie standard (*profondeur* ou *largeur d'abord*) ou définie par l'utilisateur selon des critères propres au domaine d'application (*meilleur d'abord*).

Les schémas Le langage de schémas de SMECI possède trois niveaux conceptuels distincts: les *catégories*, les *prototypes* et les *objets*. Nous dirons qu'un objet *est* d'une catégorie et *possède* un prototype.

Categories Les catégories jouent à peu près le rôle des traditionnelles classes des langages objet. Ces classes sont cependant non-extensibles en structure. Il n'est donc pas possible de définir des hiérarchies de catégories.

[1] Système Multi Expert de Conception en Ingéniérie

Une catégorie est décrite par l'ensemble de ses champs. Ces champs sont typés et définissent les propriétés caractéristiques de la catégorie et les liens qu'un de ses objets pourra avoir avec d'autres objets. Tous les objets sur lesquels le système raisonne appartiennent à une catégorie et ils héritent de tous ses champs.

Afin d'éviter de se perdre en conjectures d'ordre technique, nous allons choisir comme base de nos exemples un domaine d'application neutre. On imagine donc que l'on veut gérer l'agencement du mobilier d'un ensemble de bureaux. Le concepteur du système expert voudra créer une catégorie **Meuble**. Un meuble sera par exemple caractérisé par sa hauteur, son prix, son style et le bureau auquel il appartient.

```
Categorie Meuble

   hauteur          type: reel
   prix             type: segment
   style            type: un parmi [Louis-XVI , Directoire , Art-Deco]
   emplacement      type: objet de categorie Bureau
```

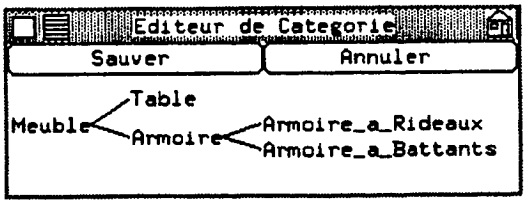
Une autre catégorie que nous manipulerons à travers nos exemples est la catégorie **Bureau** caractérisée par sa surface, l'ensemble des meubles qu'il comprend et par le coût total de cet ameublement.

```
Categorie Bureau

   surface          type: entier
   elements_mobilier type: liste d'objets de categorie Meuble
   cout_mobilier    type: entier
```

En plus des champs décrits par l'utilisateur, une catégorie naît avec la description d'un certain nombre de champs communs à toutes les catégories. Ces champs sont principalement *nom*, *existence* et *prototype*. Ils représentent, pour un objet, son nom qui doit être unique dans une base d'objets donnée, son existence dans la base d'objets courante du système expert et le prototype de cet objet. Nous décrivons ci-dessous ce qu'est un prototype.

Prototypes Les catégories sont raffinées en hiérarchies de sous-classes, non extensibles en structure, appelées prototypes. Cette hiérarchie forme ce qu'on appelle l'*arbre de prototypes* associé à la catégorie. L'arbre de prototypes de la catégorie **Meuble** est, par exemple, le suivant:



Alors que la catégorie décrit la structure des instances, les prototypes permettent de définir de la connaissance associée à la structure, sous forme de valeurs par défaut ou de contraintes de domaine. Ainsi, dans un prototype, on peut spécifier pour chaque attribut de la catégorie:

- l'intervalle de variation des valeurs numériques.

- les listes de valeurs possibles.
- la catégorie des objets, quand le champ relie l'objet à d'autres objets.

Le prototype **Armoire-a-Rideaux**, qui représente un meuble résoluement moderne, peut par exemple définir une restriction sur le prix maximum et les styles possibles.

```
Prototype Armoire-a-Rideaux
   categorie      Meuble
   prix-max       5000
   style          [Art-Deco]
```

Les prototypes permettent également d'effectuer des attachements procéduraux, dépendant de leur position dans l'arbre des prototypes. Un attachement procédural est une méthode que l'on déclenche par l'envoi d'un message à un objet. Tout objet hérite des méthodes attachées aux ancêtres de son prototype, à moins que celui-ci ne les redéfinisse localement.

Les arbres de prototypes sont munis d'un mécanisme automatique de gestion de cohérence des contraintes de domaine, où chaque prototype hérite de l'intersection des contraintes de son père d'une part et de l'union des contraintes de ses descendants d'autre part.

Objets Un objet appartient à une catégorie indiquée à sa création qui définit sa structure. Il possède un prototype qui définit le domaine de variation de chacun de ses champs et les messages auxquels il sait répondre. Les objets sont organisés en réseau par l'intermédiaire de leurs champs de type objet.

Les règles Les règles SMECI s'expriment dans une syntaxe agréable, proche de la langue naturelle et sont compilées en des objets SMECI de catégorie **Regle**. Elles comprennent les deux parties classiques: *prémisses* et *conclusions*.

Premisses Elles décrivent les conditions d'applicabilité de la règle. Elles comprennent une partie *déclaration* dans laquelle on précise, au moyen de variables typées, les objets à instancier puis une suite de prédicats sur les champs de ces objets. Ces prédicats appartiennent à un sous-ensemble de la logique du premier ordre: on peut spécifier des quantificateurs existentiels et universels. On peut par ailleurs utiliser n'importe quel prédicat LISP portant sur les objets ou sur leurs champs

Les prémisses de la règle ci-dessous instancient une armoire à rideaux appartenant au mobilier du *Bureau-102* et dont le prix est supérieur à 1500 francs.

```
defregle cherchez-l-armoire
  soit *b un Bureau
    et *a un Meuble de prototype Armoire-a-Rideaux
          parmi les elements_mobilier de *b
  si le nom de *b = Bureau-102
  et le prix de *a > 1500
alors ...
```

Conclusions Elles permettent de

- créer de nouveaux objets.
- supprimer des objets existants.

- déterminer ou modifier les valeurs des champs des objets instanciés dans la partie prémisses et notamment de
  - resserrer les contraintes réelles sur un objet.
  - préciser davantage le prototype d'un objet, c'est à dire le faire descendre dans la hiérarchie de prototypes.
- effectuer n'importe quelle action en Le-Lisp (impression de message, question à l'utilisateur, appel de routines externes, etc...)

La règle suivante a pour rôle de mettre une armoire disponible dans un bureau qui n'en a pas.

```
defregle attribuer-rangement
  soit *b un Bureau
    et *a une Armoire
  si quelque soit *m un des elements_mobilier de *b
     ona prototype de *m <> Armoire
  et l'emplacement de *a = ()
  alors
  *a est un des elements_mobilier de *b et
  l'emplacement de *a = *b
fin-regle
```

Le raisonnement Le moteur d'inférence de SMECI développe un arbre d'*états* qui représente le raisonnement accompli par le système expert. Cet arbre est parcouru selon une *stratégie* définie par l'utilisateur. Le contrôle de l'application des règles est géré par un interprète de *tâches*.

état Un état est un objet SMECI qui décrit les modifications de la base d'objets induites par l'application d'une règle sur une certaine liste d'instanciations. Ces modifications (ou *transitions*) sont des créations ou des suppressions d'objets et des affectations de valeurs à des champs d'objets. Groupés au sein d'une structure d'arbre, les états figurent le résultat d'un raisonnement. Les feuilles de l'arbre d'états sont des états dits *singuliers*. Ils correspondent à l'arrêt d'une branche de raisonnement. Cet arrêt peut avois plusieurs causes, parmi lesquelles:

- l'état a été reconnu solution du problème par une règle.
- l'état a été reconnu contradictoire par une règle.
- l'état a subi une affectation incohérente (valeur de type erroné ou hors d'un intervalle).

stratégie La stratégie d'un système expert va piloter son raisonnement. C'est un objet SMECI auquel on rattache un certain nombre d'informations qui vont guider la recherche de solutions. Ces informations sont principalement les suivantes :

- Le nombre maximal d'états que pourra créer le système expert au cours de son raisonnement.
- le temps CPU maximal accordé au déroulement du raisonnement.
- Un test de duplication qui permet de reconnaitre des états isomorphes, évitant ainsi de multiplier inutilement les recherches.
- Une fonction d'évaluation qui permet d'attribuer une note à un état dans le cas où on effectue une recherche de type "meilleur d'abord".

tâche L'expression d'un problème à résoudre en SMECI passe par sa décomposition en sous-problèmes indépendants. Ces sous-problèmes sont modélisés par des tâches. Une tâche est un objet SMECI qui est principalement caractérisé par sa *base de règles*. Cette base de règles est un objet décrivant l'ensemble des règles relatives à un sous-problème ainsi que leur utilisation. La structure des tâches permet de les organiser en un graphe qui représente leur enchaînement lors de la résolution du problème.

Un cycle de base du moteur d'inférence se déroule alors schématiquement de la façon suivante:

- Choix du meilleur état parmi les états non explorés.
- Activation de la tâche courante sur cet état. Ceci se traduit par le choix des règles et des instanciations utilisables selon la base de règle associée à la tâche puis par la création de nouveaux états par application des conclusions des règles sur les instanciations sélectionnées.
- Evaluation des états générés suivant la stratégie utilisée.

2.3.2 Expression des contraintes

Afin de pouvoir manipuler nos contraintes de façon dynamique en cours de raisonnement, nous avons étendu la grammaire initiale du langage de règles. La syntaxe de la partie *conclusions* s'est enrichie de la règle < *contraindre* > telle que:

< *contraindre* > ::= **etablir** < *type_de_contrainte* > (< *liste_d_objets* >)

Ce léger sucre syntaxique va permettre une expression purement déclarative des contraintes désirées. De plus, il correspond bien à l'esprit généralement déclaratif des règles d'inférence. La sémantique d'un tel énoncé au sein d'une règle d'inférence est la suivante:

> *Si les premisses de la règle sont valides, alors créer une contrainte liant les champs des objets de* < *liste_d_objets* > *et dont la sémantique est traduite par* < *type_de_contrainte* >.

La règle suivante utilise le type de contrainte **meme-hauteur** pour exprimer que si deux tables sont dans la même pièce, alors, par souci d'uniformité, elles doivent avoir la même hauteur.

```
defregle uniformiser
 soit *t1 un Meuble de prototype Table et
      *t2 un Meuble de prototype Table
 si l'emplacement de *t1 = l'emplacement de *t2
 alors
 etablir meme-hauteur (*t1 *t2)
fin-regle
```

2.3.3 Types de contraintes

Les types de contraintes désignés dans l'énoncé < *contraindre* > sont définis séparément à l'aide de la primitive **defcontrainte**. Cette primitive permet de définir un modèle de contrainte par la donnée d'un prédicat exprimant la relation et de l'ensemble des méthodes utilisables pour satisfaire cette relation. La syntaxe de cette primitive est [2]:

[2]N.B: Certaines < *expression* > admettent un format infixe.

< *defcontrainte* > ::= **defcontrainte** < *nom* >
soit {< *declaration* >}+
predicat < *expression* >
[**methodes** {< *methode* >}+]
fin-contrainte

< *declaration* > ::= < *variable* > { **un** | **une** } < *categorie* >

< *methode* > ::= " [" {< *reference_de_champ* >}+ "] <- " < *expression* >

La sémantique d'une méthode est que, pour satisfaire la contrainte, il suffit d'affecter respectivement les champs d'objets référencés en partie gauche avec les valeurs de la liste rendue par l'évaluation de < *expression* >. Par exemple, pour définir le type **meme-hauteur** qui permettra de maintenir l'égalité entre les hauteurs de deux objets donnés de catégorie **Meuble**, on écrira:

```
defcontrainte meme-hauteur
soit *m1 un Meuble et
     *m2 un Meuble
predicat hauteur de *m1 = hauteur de *m2
methodes [hauteur de *m1] <- hauteur de *m2
         [hauteur de *m2] <- hauteur de *m1
fin-contrainte
```

Les méthodes ci-dessus expriment que si la hauteur d'un des meubles est changée, la hauteur de l'autre doit être affectée de la même valeur. La relation entre **hauteur de *m1** et **hauteur de *m2** est ainsi non-directionelle. Le type défini ci-dessous permettra de contraindre tout objet possédant un champ **hauteur** à conserver cette hauteur dans un intervalle ouvert spécifié.

```
defcontrainte intervalle
soit *o un Objet et
     *borne-inf un entier et
     *borne-sup un entier
predicat hauteur de *o > *borne-inf et
         hauteur de *o < *borne-sup
fin-contrainte
```

On peut noter que ce type ne possède pas de méthodes. Par conséquent, les contraintes qu'il engendrera pourront être testées mais pas rétablies. La compilation de la définition d'un type de contrainte par **defcontrainte** donne naissance à un objet SMECI de catégorie **Relation**. Cette catégorie est propre au système de contraintes. Elle est caractérisée schématiquement par les champs suivants:

```
Categorie Relation

   predicat          type: fonction
   correcteur        type: fonction
```

La sémantique des champs de cette catégorie et l'utilisation qui est faite des objets **Relation** sont expliqués dans la section suivante.

2.3.4 Instances de contraintes

L'introduction de contraintes ne doit pas entrainer l'ajout d'un nouveau formalisme de représentation de connaissance au niveau de l'implantation du noyau. On doit veiller en effet à ne pas alourdir ce dernier et essayer de profiter au maximum des fonctionnalités qu'il offre sur les formalismes existants. Nos contraintes sont donc représentées par des objets standard de catégorie **Relieur**. La description de cette catégorie est la suivante :

```
Categorie Relieur

   dependances        type: liste d'objets de categorie Attribut
   relation           type: objet de categorie Relation
```

- Le champs *dependances* rassemble tous les champs d'objets impliqués par la contrainte. Chaque champ est désigné à l'aide d'un objet de catégorie **Attribut**[3]. La description de cette catégorie est la suivante :

```
Categorie Attribut

   champ              type: symbole
   objet              type: objet de categorie quelconque
   contraintes        type: liste d'objets de categorie Relieur
```

  Le champ *contraintes* d'un attribut contient tous les relieurs dans lesquels cet attribut intervient. Les réseaux de contraintes sont donc modélisés par un graphe biparti dont les noeuds sont alternativement des objets de catégorie **Relieur** et **Attribut**.

- Le champ *relation* pointe vers l'objet de catégorie **Relation** qui définit la relation à maintenir entre les champs de la liste de dépendances. La sémantique d'une relation s'exprime à travers deux champs :
  - Le champ *predicat* désigne une fonction qui, appliqué à la liste de dépendances d'un relieur retourne un booléen qui indique si la contrainte est satisfaite ou non.
  - Le champ *correcteur* désigne une fonction qui, appliquée à un ensemble de dépendances modifiées, retourne une liste de couples (*attribut.valeur*) représentant la liste des ajustements à effectuer pour satisfaire localement la contrainte.

2.3.5 Création des contraintes

L'exécution d'un énoncé < *contraindre* > lors de l'application d'une règle se traduit par :

- la création d'un objet **Relieur**
- la création des objets **Attribut** qui n'existent pas encore
- l'établissement des liens entre attributs et relieur
- l'établissement du lien entre le relieur et sa relation

[3] Cet artifice est inutile quand on travaille dans un langage de schéma dans lequel les champs d'objets sont eux-mêmes des objets

On peut voir la déclaration de contraintes dans les règles comme une macro-définition du langage de règles qui sera expansée en la suite d'actions décrite ci-dessus. Le fait que ces actions se conforment à celles couramment autorisées dans les conclusions des règles garantit d'une certaine façon l'innocuité de la déclaration vis-à-vis du raisonnement du système expert. Le réseau des contraintes s'étend ainsi incrémentalement dans la base d'objets, au fur et à mesure de l'application de règles contraignantes.

2.3.6 Maintien de la cohérence

Après toute création d'un nouvel état par l'application d'une règle, nous devons assurer que celui-ci est cohérent eu égard à son ensemble de contraintes. Nous savons que l'état père de l'état considéré était cohérent puisqu'il a pu générer un fils. Il nous suffit donc de considérer l'ensemble des transitions propres à l'état. On en extrait tous les relieurs créés ainsi que tous les attributs correspond à une des transitions. Ces deux ensembles sont alors passés à l'algorithme de propagation décrit en 1.4.2. Le résultat de la propagation est soit un ensemble d'affectations soit la déclaration d'une contradiction si les modifications apportées par la règle sont incohérentes. Dans le premier cas, les affectations sont effectuées et leur description est ajoutée aux transitions de l'état. Dans le second cas, l'état examiné est déclaré singulier. La branche de raisonnement du système expert s'arrête alors sur cet état.

2.4 Conclusion

La représentation des contraintes dans le formalisme des objets du noyau présente de nombreux avantages. Elle profite notamment du mécanisme de retour arrière dans l'arbre de raisonnement, c'est à dire qu'une contrainte créée dans un état quelconque n'existe plus lorsqu'on remonte sur un de ses état ancêtres. Néanmoins, un problème survient lorsqu'un utilisateur retors vient à essayer de contraindre les champs d'un relieur ou d'une relation. A priori, ceci est tout à fait légal mais il devient alors impossible de garantir l'intégrité de la sémantique des contraintes. Prenons par exemple le cas d'une contrainte C_{meta} portant sur la liste de dépendances d'une autre contrainte C_{objet}. L'algorithme de propagation ne définit pas de précédence entre le calcul des dépendances d'une contrainte et l'utilisation de ces dépendances. Il pourrait donc arriver qu'une des variables de C_{objet} soit recalculée sur la base d'une certaine liste de dépendances puis qu'au cours des ajustements suivants, la valeur de cette liste soit modifiée par le déclenchement de C_{meta}. L'état résultant serait alors incohérent. Une solution simple et radicale pour éviter ces problèmes est d'interdire l'expression de contraintes impliquant des champs d'objets système. Une approche plus délicate consiste à insérer dans l'algorithme de propagation un traitement *ad hoc* pour les champs d'objets système. Dans tous les cas, la gestion de restrictions ou d'exceptions nous semble aller à l'encontre du principe d'uniformité qui guide (ou devrait guider) tout développement de sous-systèmes en SMECI. D'autre part et ainsi que nous l'enseignent les préceptes fondamentaux du génie logiciel, l'implantation de tels cas particuliers implique souvent sur le code une augmentation de sa taille, de sa complexité, le verminage de parties initialement saines et des difficultés de maintenance. Au vu de cette liste imposante de monstruosités, la conclusion qui s'impose est que notre algorithme de propagation doit être refondu afin d'offrir un comportement uniforme et sémantiquement correct quels que soient les objets contraints.

D'autre part, une attitude saine lorqu'on est confronté à des considérations de type "traitement ad hoc" ou "activité méta" au cours de l'élaboration d'un système est de penser à une *architecture réflexive*. L'objet du chapitre suivant est précisément d'exposer l'impact de la réfléxivité sur notre système de contraintes.

3 Vers un système réflexif

Dans ce chapitre, nous abordons le concept de réflexivité appliqué aux systèmes informatiques. Ce concept s'avère assez délicat à appréhender; en particulier, la frontière au delà de laquelle un système mérite d'être qualifié de réflexif est souvent mal perçue. En revanche, il apparait clairement que les approches réflexives permettent de résoudre de façon élégante des problèmes inhérents à l'organisation de l'activité du système. Outre des considérations d'ordre esthétique, nous voulons montrer que la prise en compte du concept de réflexivité lors de l'élaboration d'un système à base de contraintes confère à celui-ci un gain important en matière d'expressivité et de souplesse.

3.1 Généralités sur les systèmes réflexifs

Un système exécute sur un domaine les actions décrites par un programme. Le système arbore un comportement réflexif lorsqu'il se prend lui-même pour domaine d'activité, c'est à dire qu'il applique des actions à une représentation de son état. L'activité réflexive ne contribue donc pas directement à la résolution d'un problème puisqu'elle n'agit pas sur les objets du domaine du problème. En revanche, elle a pour but de faciliter et d'adapter l'activité du système face aux problèmes qui lui sont soumis.

L'intégration du concept de réflexivité passe par la détermination d'un *modèle* pour un certain nombre de composants du système. Cette détermination s'effectue relativement à la théorie que l'on a des éléments du niveau objet [Bat83]. Le modèle va fixer l'ensemble des activités que le système peut accomplir sur lui-même; celui-ci sera plus ou moins adapté à une activité réflexive donnée. Il n'existe donc pas de modèle complet ou optimal. La plupart des systèmes réflexifs n'offrent qu'une seule représentation d'eux-mêmes. [Mae86] pose les bases d'un langage orienté-objet réflexif offrant plusieurs points de vue sur un même programme.

Pour que la réflexivité soit intéressante, il faut qu'il existe un *lien de causalité* entre l'activité du système et sa représentation. Ce lien de causalité garantit que l'activité et sa représentation sont toujours en accord, c'est à dire que la modification de l'une entraine une modification correspondante de l'autre. Par exemple, une jauge graphique sur un écran est associée à une variable représentant la quantité jaugée. Il existe un lien de causalité entre ces deux objets si la modification de la quantité déclenche un réaffichage de la jauge et si une action sur la jauge à l'aide de la souris affecte une nouvelle valeur à la quantité jaugée. Ainsi, à l'issue d'une activité réflexive, la reprise de l'activité de niveau inférieure se trouve effectivement affectée par les modifications de sa représentation. Un système qui offre uniquement la possibilité d'observer son comportement sans pouvoir le modifier ne peut pas être qualifié de réflexif [Mae86].

Une fois le modèle du système déterminé et relié à ce qu'il modélise, il reste encore à déterminer quand et comment utiliser les capacités réflexives du système. A un niveau donné, il faut pouvoir arrêter l'activité du système, passer à un niveau méta puis retourner au niveau objet après avoir manipulé sa représentation. Une architecture réflexive doit donc définir un *contrôle de l'activation du processus réflexif.*

[Fer87] distingue trois type d'approches réflexives selon le type d'activité du système et les connaissances utilisées au cours de son activité réflexive. On trouve ainsi une réflexivité *structurale*, *procédurale* et *conceptuelle*. Nous présentons plus en détail la réflexivité procédurale car notre système de contraintes entre dans ce cadre.

3.1.1 Réflexivité procédurale

Un langage dont les programmes ont la possibilité d'accéder à une représentation d'eux-mêmes et d'intervenir sur le déroulement de leur processus d'exécution est doué d'un comportement réflexif. Par son uniformité de représentation entre les données et les programmes, LISP offre un terrain propice à l'introduction du concept de réflexivité. En effet, LISP permet d'exécuter des données et de manipuler des programmes. D'autre part, la plupart de ses dialectes donnent accès à leur environnement par des fonctions telles que `boundp`[1] ou à leur pile d'exécution par `cstack`[1], `tag`[1] ou `exit`[1]. Ces fonction sont souvent utilisées pour construire des outils de trace ou d'exécution incrémentale (pas-à-pas). Néanmoins, l'environnement d'exécution n'est pas explicitement représenté par des objets du langage.

3-LISP [Smi82] est un dialecte réflexif de LISP dans lequel une fonction réflexive accède à l'environnement d'exécution de la fonction qui l'a invoquée. Cet environnement est représenté par des variables LISP : l'ensemble des variables liées est donné sous forme d'une a-liste et la continuation par une s-expression. Le lien de causalité entre modèle et activité existe naturellement en 3-LISP du fait de l'utilisation d'un interprète méta-circulaire [Cou78]. Un interprète méta-circulaire utilise en effet une représentation explicite de l'interprétation pour accomplir effectivement cette interprétation. Enfin, l'activité réflexive d'un tel système est explicitement contrôlée par les programmes eux-mêmes c'est à dire que le code lui-même comporte des appels aux fonctions réflexives.

3.1.2 Apports des approches réflexives

De nombreuses activités d'un système informatique sont réflexives par essence c'est à dire qu'elles nécessitent l'accès à une représentation du programme ou de son état. Ces activités peuvent ne faire intervenir que des concepts intérieurs au système. C'est le cas, par exemple, des fonctions de trace et d'exécution pas-à-pas. D'autres impliquent l'intervention de connaissances extérieures au système. Parmi celles-ci, on trouve l'apprentissage ou bien le raisonnement sur le contrôle. Ainsi, un système qui apprend doit être capable de s'auto-modifier à la suite d'inférences qu'il vient d'accomplir sur un problème quelconque. Ces modifications font évoluer son comportement ultérieur confronté à un nouvel exemplaire du même problème. Evidemment, il est essentiel de procurer au système un critère permettant de juger le bien-fondé de ces auto-modifications afin qu'il progresse.

[Cor88] dégage certains avantages des approches réflexives du point de vue génie logiciel dans le domaine de l'intelligence artificielle. Ces avantages apparaissent aussi bien du côté du développeur que de l'utilisateur du système. Tout d'abord, la programmation d'un système dans le même formalisme que celui des connaissances peut amener une réduction importante de la taille du code du système. On limite de toutes façons le nombre d'applications utilitaires à écrire comme les gestionnaires d'entrée-sortie, les outils d'édition ou de trace. Dans le même temps, l'utilisateur n'est pas obligé d'assimiler autant de formalismes que de concepts. Un système réflexif apparaît aussi plus facilement extensible ou modifiable. En effet, dès lors que l'on maitrise bien le formalisme de représentation des connaissances, on peut se lancer dans des opérations de chirurgie et d'extension de la méta-connaissance.

3.2 Réflexivité et contraintes

Nous avons vu dans le chapitre précédent que la représentation de nos contraintes permet d'exprimer naturellement des méta-contraintes. Or notre algorithme s'avère incapable de gérer correctement de telles contraintes. On peut alors soit penser que l'algorithme n'est pas approprié, soit

[1] Fonctions du dialecte Le_Lisp [Cha87].

douter du bien fondé de la représentation des contraintes. La section suivante montre clairement sur des exemples l'utilité de méta-contraintes. Le choix de la représentation étant justifié, nous montrons alors l'influence de telles contraintes sur l'algorithme de satisfaction.

3.2.1 Un cas pathologique

Le problème que nous allons décrire se pose lorsqu'on désire représenter des contraintes dans une base d'objets dont l'organisation est extrèmement dynamique. On suppose qu'un système expert travaille sur des objets dont les catégories sont décrites en 2.3.1. La base d'objets contient un objet $\mathcal{B}$ de catégorie **Bureau** et n objets $\mathcal{M}_1 \ldots \mathcal{M}_n$ de catégorie **Meuble**. Le champ *elements_mobilier* de $\mathcal{B}$ peut contenir toute combinaison de l'ensemble $\mathcal{M}_1 \ldots \mathcal{M}_n$. A tout moment, au cours du raisonnement, des objets de catégorie **Meuble** peuvent être ajoutés ou retirés de la liste *elements_mobilier*($\mathcal{B}$). De plus, le champ *prix* de chaque objet de catégorie **Meuble** peut lui aussi être re-évalué par le système expert. Nous voulons à présent établir la contrainte suivante qui exprime de façon naturelle que le coût total de l'ameublement d'un bureau est la somme du prix de tous les meubles composant cet ameublement. Cette contrainte s'exprime de la façon suivante :

$$cout\_mobilier(\mathcal{B}) = \sum_{\mathcal{M}_i \in elements\_mobilier(\mathcal{B})} prix(\mathcal{M}_i) \tag{3.1}$$

On suppose enfin qu'au moment où on établit la contrainte, on a:

$$elements\_mobilier(\mathcal{B}) = (\mathcal{M}_1, \mathcal{M}_3, \mathcal{M}_7)$$

Que devons nous faire pour exprimer la contrainte 3.1 dans un langage de contraintes classique à base de dépendances ? Deux approches se présentent spontanément, qui vont se réveler fausses toutes les deux. La première consiste à créer une dépendance entre les champs *elements_mobilier* et *cout_mobilier* de l'objet $\mathcal{B}$. Cette dépendance permettra d'actualiser *cout_mobilier*($\mathcal{B}$) chaque fois que *elements_mobilier*($\mathcal{B}$) sera modifié. Par exemple, le fait d'ajouter le meuble $\mathcal{M}_2$ au mobilier de $\mathcal{B}$ va déclencher le calcul de la nouvelle valeur de son champ *cout_mobilier*. Mais que se passera-t-il si le prix de $\mathcal{M}_1$ est modifié au cours des inférences futures du système expert? Comme il n'existe pas de dépendances entre le champ *prix* de $\mathcal{M}_1$ et le champ *cout_mobilier* de $\mathcal{B}$, la contrainte ne sera pas réveillée et par conséquent, elle restera dans un état incohérent. Qu'à cela ne tienne. Nous allons prendre le problème sous un autre angle et créer une dépendance entre chaque champ *prix* des objets dans *elements_mobilier*($\mathcal{B}$) et le champ *cout_mobilier* de $\mathcal{B}$. Le problème est alors reporté sur le champ *elements_mobilier* de $\mathcal{B}$ car si celui-ci est modifié, le champ *cout_moblier* de $\mathcal{B}$ ne sera pas mis à jour. En désespoir de cause, on peut penser à ajouter le champ *elements_mobilier* de $\mathcal{B}$ à l'ensemble des dépendances de la contrainte. Le problème n'est pas résolu pour autant puisque l'ajout ou le retrait d'un meuble dans *elements_mobilier* ne s'accompagnera pas d'une mise à jour en conséquence de l'ensemble des dépendances.

Au-delà d'une simple combinaison, nous ressentons donc le besoin d'une interaction entre les deux types de dépendances que nous venons de présenter. Ainsi, afin de rester cohérent dans un environnement dynamique, des contraintes telles que 3.1 doivent être gérées explicitement par l'utilisateur. Cela veut dire que, quand un lien de dépendance doit être ajouté ou retiré, la contrainte qui possède ou nécessite ce lien devient obsolète. L'utilisateur doit alors la retirer et la remplacer par une nouvelle[2]. Nous prétendons que cette obligation n'est pas acceptable dans le cadre d'environnements aussi dynamiques que les bases d'objets des systèmes experts.

[2]Ce problème est une généralisation de celui évoqué dans les conclusions de la thèse de Steele ([Ste80] 10.2.1). Il propose l'implantation d'un système dans lequel les contraintes pourraient être créées sans que toutes ses dépendances soient connues.

En effet, au cours des premières phases du raisonnement, les objets de la base sont sujets à de fréquents changements au niveau de leurs sous-parties, reflétant ainsi la recherche d'une alternative de conception par exemple. Ceci ne doit en aucun cas être préjudiciable au maintien de la cohérence. L'utilisateur doit pouvoir exprimer des contraintes de manière abstraite et laisser au système de contraintes l'initiative de la gestion des dépendances. Pour satisfaire ceci, nous pouvons remarquer que le fait de dire que l'ensemble des dépendances de la contrainte est égale à l'ensemble des champs *prix* de tous les meubles de $elements\_mobilier(\mathcal{B})$ n'est rien d'autre qu'une contrainte. Cette contrainte agit au niveau *méta* puisqu'elle contraint une partie d'une autre contrainte.

3.2.2 Une solution avec méta-contraintes

Pour la clarté de l'exposé, nous introduisons ici quelques notations. On notera $\&(c, \mathcal{O})$ l'objet de catégorie `Attribut` qui représente le champ c de l'objet $\mathcal{O}$. On définit par ailleurs une fonction $\star$ de déréférençage des attributs telle que :

$$\star(\&(c, \mathcal{O})) = c(\mathcal{O})$$

Muni de ces conventions, on peut aisément exprimer la contrainte 3.1 par deux contraintes: une au niveau objet et l'autre au niveau méta. La contrainte de niveau objet est représentée par le relieur $\mathcal{C}_1$ et telle que :

$$cout\_mobilier(\mathcal{B}) = \sum_{\mathcal{D} \in dependances(\mathcal{C}_1)} \star(\mathcal{D})$$

Cette contrainte va assurer que le coût du mobilier est la somme de la valeur de ses dépendances. Ces dépendances sont les attributs représentant les champs *prix* des meubles de la liste $elements\_mobilier(\mathcal{B})$. Au moment où la contrainte est exprimée, ces dépendances sont $\&(prix, \mathcal{M}_1)$, $\&(prix, \mathcal{M}_3)$ et $\&(prix, \mathcal{M}_7)$. D'autre part et afin de prendre en compte l'évolution de cette dernière, on doit établir la méta-contrainte sur $\mathcal{C}_1$ qui sera représentée par le relieur $\mathcal{C}_2$, telle que :

$$dependances(\mathcal{C}_1) = \bigcup_{\mathcal{M}_i \in elements\_mobilier(\mathcal{B})} \&(prix, \mathcal{M}_i)$$

Ces deux contraintes sont exprimées une fois pour toutes et vont permettre de maintenir la contrainte 3.1 quels que soient les changements susceptibles d'intervenir sur le champ *elements_mobilier* de $\mathcal{B}$ ou sur le *prix* des $\mathcal{M}_i$.

3.2.3 Autres applications

On se rappelle le problème concernant la référence de variables par des chemins à travers des hiérarchies de sous-parties, évoqué en 2.2. Ce problème se ramène aussi à la gestion de l'ensemble des dépendances d'une contrainte et peut donc se résoudre à l'aide de méta-contraintes. Supposons par exemple que l'on veuille établir la contrainte

$$surface(emplacement(\mathcal{M}_6)) > 12$$

Nous allons exprimer cette contrainte à l'aide de deux relieurs. Le relieur $\mathcal{C}_3$ représente la contrainte qui assure que la valeur de sa dépendance est supérieure à 12 :

$$\star(dependances(\mathcal{C}_3)) > 12$$

Cette dépendance est la surface de l'emplacement de $\mathcal{M}_6$. Elle doit être modifiée selon l'emplacement du meuble. Cette modification est assurée par une seconde contrainte, représentée par un relieur C_4, telle que :

$$dependances(C_3) = \&(surface, emplacement(\mathcal{M}_6))$$

Bien sûr, ceci est généralisable pour un chemin de longueur n et on peut envisager de générer automatiquement la description des méta-contraintes. On obtient ainsi un système beaucoup plus uniforme que celui réalisé grâce à des attachements procéduraux par SOCLE [Har86].

Jusqu'ici, nous avons montré l'utilité de contraindre le champ *dependances* d'une contrainte. Nous étudions à présent quel peut être l'intérêt de contraindre ses autres champs.

Chaque objet possède un champ *existence* (c.f. 2.3.1) qui définit son existence dans la base d'objets courante. "$existence(\mathcal{O}) = ()$" indique que $\mathcal{O}$ n'est pas présent dans la base d'objets. Le fait de contraindre le champ *existence* d'un objet **Relieur** va permettre d'exprimer des *contraintes conditionnelles*. Une contrainte conditionnelle est une contrainte dont l'existence est subordonnée à la validité d'une condition. Elle doit donc cesser d'être active lorsque sa condition n'est plus vérifiée et être redéclenchée dès que cette condition redevient valide. Ainsi, supposons que deux tables $\mathcal{T}_1$ et $\mathcal{T}_2$ aient été liées par la contrainte **meme-hauteur** lors d'une application de la règle **uniformiser** décrite en 2.3.2. Soit C le **Relieur** représentant cette contrainte. Nous voulons que cette contrainte ne soit effectivement prise en compte que lorsque les deux tables appartiennent au même bureau. Pour cela, il nous suffit d'établir une méta-contrainte portant sur le champ *existence* de C telle que:

$$existence(C) = eq(bureau(\mathcal{T}_1), bureau(\mathcal{T}_2))$$

Dès que $\mathcal{T}_1$ et $\mathcal{T}_2$ ne seront plus dans le même bureau, l'existence de C sera () et la contrainte cessera de peser sur les hauteurs des tables. Réciproquement, lorsque les deux tables seront remises dans le même bureau, C redeviendra active et les hauteurs des deux tables seront réajustées si nécessaire.

Bien qu'il soit aussi possible de piloter la valeur du champ *relation* d'une contrainte, nous ne voyons pas l'intérêt, dans des applications usuelles des contraintes, de modifier dynamiquement la relation existant entre un ensemble d'attributs donné.

3.2.4 Satisfaction avec méta-contraintes

Nous allons voir dans cette section que les contraintes réflexives, si elles apportent beaucoup de souplesse et de simplicité au niveau de l'expression, nécessitent un algorithme de satisfaction qui n'est pas aussi direct que l'algorithme classique.

N.B. : Les remarques que nous exprimons ici n'ont rien de formel et restent très intuitives.

Retour arrière lors de la construction d'une alternative

Comme nous l'avons exposé au chapitre précédent, le centre de notre système de satisfaction de contraintes est un algorithme basé sur le principe de propagation. Ce principe peut être vu comme un ordre particulier pour le parcours d'un graphe, à l'instar de la profondeur d'abord ou de la largeur d'abord [Per87]. L'ordonnancement est basé sur des informations purement locales extraites des noeuds du graphe. Que le parcours se fasse en une phase avec calcul en chaque noeud ou en deux phases par planification puis recalculs, il n'est jamais remis en cause par une exploration plus en avant.

Lorsque l'on s'attache à satisfaire des méta-contraintes, le parcours du graphe peut impliquer des modifications de sa topologie puisque les ensembles de dépendances ou l'existence même

des contraintes, et donc des noeuds du réseau, est calculée dynamiquement. La construction des graphes de propagation que nous avions décrite en 1.4.2 va à présent se faire par une stratégie d'essai-erreur assistée par retour arrière. En effet, on doit remettre en cause cette construction lorsque l'on modifie dynamiquement une partie de la structure du réseau de contraintes après avoir déjà parcouru cette même partie. Explicitons les occurences précises et les conséquences de ce retour arrière.

Au cours de la construction d'un graphe de propagation, l'algorithme procède pour ses propres besoins à la lecture de la valeur de certains des attributs des contraintes. Il faut en effet s'assurer qu'une contrainte existe avant de la traiter puis déterminer quelles sont ses dépendances pour continuer à propager. Ces actions impliquent respectivement la lecture du champ *existence* et *dependances* du relieur représentant la contrainte. On peut donc considérer que les attributs utilisés pour la construction d'un noeud sont des antécédents d'*expansion* de ce noeud, c'est à dire que leur valeur doit être connue avant que le noeud puisse être expansé[3]. Si l'antécédent d'expansion dont on requiert la valeur figure comme conséquent d'un noeud du graphe de propagation, sa valeur courante ne peut être utilisée immédiatement. Il faut d'abord lancer son évaluation afin de connaître sa nouvelle valeur. Cette évaluation entraine l'évaluation récursive de la fermeture transitive des antécédents du noeud considéré comme on l'a vu en 1.4.2. Le graphe peut donc comporter, avant sa complétion, des attributs calculés et dont la valeur a été utilisée.

Il peut alors arriver que l'algorithme détruise la correction causale du graphe de propagation soit en ajoutant un antécédent à un noeud déjà évalué soit en créant un noeud dont un des conséquent est un antécédent d'expansion d'un autre noeud du graphe. Cela correspond à la remise en cause de la valeur d'un antécédent d'expansion sur laquelle s'est basé une partie de la propagation. Ainsi et afin que la sémantique des dépendances entre les noeuds reste cohérente, il faut effectuer un retour arrière sur le graphe de propagation à partir du noeud qui a sollicité l'évaluation de l'antécédent d'expansion devenu obsolète. Ce retour arrière a pour rôle d'une part d'éliminer du graphe de propagation tous les sommets qui dépendent uniquement du noeud incriminé et d'autre part de replacer ce noeud dans l'ensemble des noeuds à expanser de l'alternative en cours de développement, ceci afin de recommencer sa propagation avec une valeur correcte de son antécédent d'expansion.

Causes de déclenchement d'une contrainte

Toujours dans les systèmes de propagation classiques, une contrainte n'est déclenchée qu'après qu'un attribut impliqué dans celle-ci est modifié. Avec notre représentation, une autre raison de réveiller une contrainte peut être la modification d'un de ses propres champs. En effet, si la valeur d'un champ d'une contrainte est modifiée au cours de la propagation, on doit considérer que c'est une nouvelle contrainte qui est installée dans la base d'objets et on doit donc déclencher son application. Un exemple naif de propagation peut illustrer ceci. Supposons qu'une règle d'inférence ajoute $\mathcal{M}_6$ à la liste $elements\_mobilier(\mathcal{B})$. La contrainte $\mathcal{C}_2$ impliquant $elements\_mobilier(\mathcal{B})$ est déclenchée la première. Elle est satisfaite en recalculant la valeur de $dependances(\mathcal{C}_1)$. La modification d'un champ de $\mathcal{C}_1$ déclenche celle-ci à son tour et va recalculer la nouvelle valeur de $cout\_mobilier(\mathcal{B})$.

[3] jusqu'à présent, les antécédents d'un noeud étaient des antécédents de *calcul* c'est à dire que leur valeur devait être connue avant de pouvoir calculer la valeur des conséquents du noeud.

3.3 Le système PROSE

Au sein du générateur SMECI, nous avons implanté notre système de satisfaction de contraintes sous la forme d'un système expert appellé PROSE[4]. Nous justifions ici ce choix et nous étudions la réalisation de ce système.

3.3.1 Réalisation

Comme son nom veut l'indiquer, le générateur SMECI permet l'écriture de systèmes **multi-experts**. Le terme multi-experts signifie ici que SMECI est capable de gérer le raisonnement de plusieurs systèmes experts indépendants et d'entrelacer leur processus en les faisant s'observer et éventuellement interagir. Ceci est initialement rendu possible par la réification[5] du raisonnement qu'effectue le moteur d'inférence (c.f. section 2.3.1). Plusieurs représentations de raisonnements peuvent ainsi cohabiter de façon simultanée. A leur tour, les systèmes experts SMECI sont des objets de catégorie SE (pour Systeme-Expert) qui développent leur propre raisonnement. Au sein de chaque système expert, le raisonnement passé, présent et futur est décrit respectivement par un *arbre d'états*, un *état courant* et un *agenda*. Outre ces descriptions, un objet de catégorie SE est caractérisé par une base de connaissance propre qui comporte la définition de la tâche à réaliser et la stratégie de raisonnement à utiliser.

La réification de la plupart des concepts de SMECI est bien sûr une porte ouverte sur l'utilisation de différentes formes de méta-raisonnement. En effet, le moteur de SMECI peut unifier sur des objets de catégorie SE et accéder à leur raisonnement en unifiant sur leurs états. Un système expert se_1 peut ainsi appliquer des règles qui vont manipuler l'état interne d'un autre système expert se_0. PROSE s'implante donc naturellement comme un système expert indépendant raisonnant sur un système observé donné. Notons que cette approche a déjà été utilisée en SMECI pour élaborer un système expert explicateur [DCH87] qui observe le système expert à expliquer.

Le raisonnement de PROSE sera lancé à l'issue de la création de chaque nouvel état du système observé et aura pour but, partant des modifications induites par l'état, de trouver le meilleur ensemble d'affectations de variables qui restaure la cohérence de la base d'objets du système observé.

3.3.2 Avantages

La motivation principale de l'écriture de PROSE est la réduction de la taille du code source ainsi que la facilité de maintenance de ce code. D'autre part, nous avons utilisé SMECI comme un langage de programmation pour la génération et le parcours d'arbres d'alternatives. Par exemple, le changement de stratégie pour la génération des alternatives de propagation se fait naturellement en changeant la stratégie du système expert. Ceci constitue une opération élémentaire en SMECI puisqu'il suffit de changer l'objet `Strategie` attaché à l'objet représentant le système expert. On profite aussi de la gestion d'états multiples réalisée par SMECI, c'est à dire que l'on n'a pas à se préoccuper de conserver des contextes pour les affectations effectuées dans une alternative puisque les affectations dans un état n'ont pas d'existence dans les états pères.

3.3.3 Implantation

Une application SMECI est programmée par une base de connaissances. Il en va de même pour le système PROSE qui définit :

[4] **PRO**pagation pour et par **S**ystème-**E**xpert

[5] Transformation d'une *idée* en une entité manipulable en tant qu'*objet* par un programme [Fer87].

- des catégories pour modéliser les noeuds du graphe de propagation ainsi que les alternatives.
- des règles qui effectuent les actions de la propagation en créant et en manipulant des noeuds et des alternatives.
- des méthodes définissant certains comportements pour les noeuds et les alternatives.
- des objets `Tache` et `Base_de_regles` qui définissent le contrôle

Le système propagateur actuel représente 10 règles réparties en 6 bases de règles et approximativement 150 lignes de code pour les méthodes. On voit donc que le code est très compact surtout si on le compare à celui de la version écrite en LE_LISP qui totalise environ 1200 lignes. Il y a bien sûr un prix à payer pour cette approche claire et concise : le temps d'exécution de la propagation se trouve légèrement augmenté.

3.4 Conclusion.

La représentation que nous avons donné de nos contraintes leur permet de s'adapter à des environnements tels que des bases d'objets complexes des systèmes experts, tout en conservant leur sémantique à travers l'évolution des liens entre objets. Cette représentation nous a amenés à considérer la notion de réfléxivité. Cette investigation doit encore être approfondie. On envisage en effet de découvrir le caractère réfléxif de l'algorithme de satisfaction. Pour cela, il nous faut définir cet algorithme comme un interprète à part entière dont les programmes interprétés seraient des liste de contraintes à déclencher. L'interprétation d'une méta-contrainte ferait alors passer l'interprète dans un état introspectif qui effectuerait le retour arrière sur le graphe (l'environnement de l'interprète) et qui modifierait la liste des contraintes à déclencher (c'est à dire la continuation). On envisage aussi d'appliquer les méta-contraintes à des activités de contrôle de l'algorithme de propagation comme cela est évoqué dans [Bor79].

A Propagation : algorithmes et exemple

A.1 Construction et évaluation des graphes de propagation

Nous décrivons ici les algorithmes évoqués en 1.4.2 pour la génération des graphes de propagation et l'évaluation d'un noeud du graphe. L'expression que nous en livrons n'est pas optimisée afin de rester lisible.

Structures Outre les variables et les contraintes du problème, l'algorithme utilise les trois structures suivantes:

- des *noeuds* comprenant les champs
  antecedents : liste de noeuds
  contrainte : contrainte
  consequents : liste de variables
  marque : booléen
- des *alternatives* comprenant les champs
  a_expanser : liste de noeuds
  expanses : liste de noeuds
- un *sac* comprenant le champ
  alternatives : liste d'alternatives

On dispose des fonctions *creer_noeud*, *creer_alternative* et *creer_sac* pour la création de ces structures. On dispose d'autre part d'une fonction *copier* pour les noeuds et les alternatives qui permet d'en dupliquer un exemplaire donné. Enfin, on peut déposer une alternative A dans le sac S par $deposer(A, S)$ et extraire une alternative du sac par $extraire(S)$.

Propagation Lors du lancement de la propagation, on regroupe dans Δ_C (resp. Δ_v) toutes les contraintes créées (resp. toutes les variables modifiées) depuis la dernière propagation. On exécute d'abord une procédure d'initialisation qui va définir et organiser l'ensemble initial des noeuds à expanser. Cette procédure construit un noeud racine qui symbolise l'action de l'utilisateur sur le réseau de contraintes. Ce noeud n'est associé à aucune contrainte et sa liste de conséquents est l'ensemble des variables modifiées c'est à dire Δ_v.

algorithme *Initialisation*(S : *sac*, Δ_C : *liste_de_contraintes*, Δ_v : *liste_de_variables*)
$A \leftarrow creer\_alternative()$,
pour-tout $C \in \Delta_C$ faire
    $N \leftarrow creer\_noeud()$,
    $contrainte(N) \leftarrow C$,
    $a\_expanser(A) \leftarrow a\_expanser(A) \cup \{N\}$,
si $\Delta_v \neq \emptyset$
alors $N \leftarrow creer\_noeud()$,
    $marque(N) \leftarrow t$
    $consequents(N) \leftarrow \Delta_v$,
    $expansion(A, N)$,
$deposer(A, S)$.

Une fois le sac initialisé, on peut lancer le développement des alternatives. L'ensemble des groupes de variables modifiables pour la satisfaction d'une contrainte C sont donnés par $modifiables(C)$.

algorithme $Propagation(S : sac)$
tant-que $alternatives(S) \neq \emptyset$ faire
$A \leftarrow extraire(S)$,
si $a\_expanser(A) = \emptyset$
alors A est une alternative complète.
sinon $N \leftarrow premier(a\_expanser(A))$,
$\mathcal{E} \leftarrow \{ \epsilon \in modifiables(contrainte(N)),$
$\forall v \in \bigcup_{aN \in antecedents(N)} consequents(aN), v \notin \epsilon\}$,
si $\mathcal{E} \neq \emptyset$
alors pour-tout $\epsilon \in \mathcal{E}$ faire
$A_\epsilon \leftarrow copier(A)$,
$N_\epsilon \leftarrow copier(N)$,
$consequents(N_\epsilon) \leftarrow \epsilon$,
$expansion(A_\epsilon, N_\epsilon)$,
$deposer(A_\epsilon, S)$,
sinon $expanses(A) \leftarrow expanses(A) \cup \{N\}$,
$deposer(A, S)$.

Dans le cas où on trouve une alternative complète, on peut soit arrêter la recherche (on a trouvé une solution potentielle) soit continuer la génération des alternatives suivantes (afin de comparer plusieurs alternatives complètes par exemple). L'expansion d'un noeud N dans une alternative A est réalisée par :

algorithme $Expansion(A : alternative, N : noeud)$
$expanses(A) \leftarrow expanses(A) \cup \{N\}$,
pour-tout $v \in consequents(N)$ faire
pour-tout $C \in contraintes\_portant\_sur(v)$ faire
si $C \neq contrainte(N)$
alors si $\exists N_c \in a\_expanser(A) \cup expanses(A)$
tel que $contrainte(N_c) = C$
alors si $N \notin antecedents(N_c)$
alors $antecedents(N_c) \leftarrow antecedents(N_c) \cup \{N\}$
sinon $N_c \leftarrow creer\_noeud()$,
$contrainte(N_c) \leftarrow C$,
$antecedents(N_c) \leftarrow \{N\}$,
$a\_expanser(A) \leftarrow a\_expanser(A) \cup \{N_c\}$.

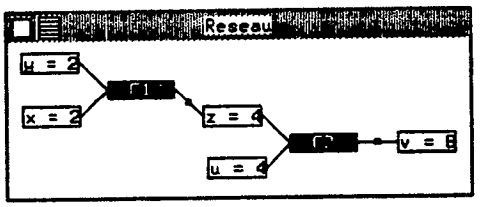
Evaluation L'algorithme de propagation ne boucle pas sur un réseau de contraintes comportant une circularité puisque chaque contrainte ne peut passer qu'une seule fois dans la liste des noeuds à expanser d'une alternative. Néanmoins, le graphe de propagation construit peut être circulaire. Il est donc nécessaire d'effectuer un test de circularité lors de l'évaluation des variables. Pour évaluer une variable, on cherche le noeud dont elle est un des conséquents sur lequel on applique l'algorithme suivant, avec une liste de noeuds déjà visités initialement nulle.

algorithme *Evaluation*(N : *noeud*, Vus : *liste_de_noeuds*)
si $\neg$*marque*(N)
alors si $N \in Vus$
alors le graphe de propagation comporte une circularité.
sinon pour-tout $N_a \in$ *antecedents*(N) faire
evaluation($N_a, Vus \cup \{N\}$),
si *consequents*(N) = ()
alors tester *contrainte*(N)
sinon calculer *consequents*(N) avec *contrainte*(N),
marque(N) $\leftarrow$ t,

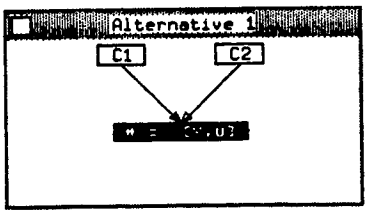
Lorsqu'on détecte une circularité lors de l'évaluation, on peut soit décider d'abandonner l'alternative soit d'appliquer une méthode numérique telle que la relaxation si la nature des contraintes s'y prête (c.f. 1.3.5). D'autre part, si le test d'une contrainte révèle une contradiction, l'alternative n'est pas une solution de la propagation.

A.2 Exemple de construction d'alternatives

Nous montrons ici un exemple du déroulement de la construction des alternatives de propagation. Pour ce faire, reprenons les deux contraintes $C_1 : z = x + y$ et $C_2 : v = z + u$ exprimées en 1.4. On rappelle le réseau qui les représente :



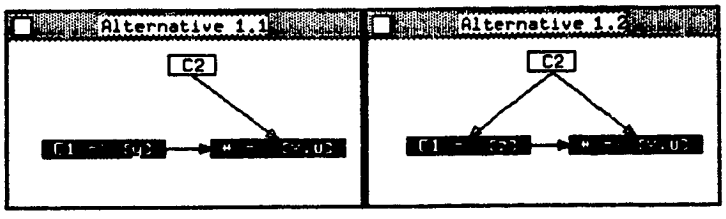
On suppose qu'on modifie initialement les variables x et u. On va donc créer un noeud racine dont les conséquents sont $\{x, u\}$ et dont l'expansion donne naissance à deux noeuds à expanser de contraintes C_1 et C_2. La pile d'alternatives contient alors l'alternative initiale 1 présentée ci-dessous[1].



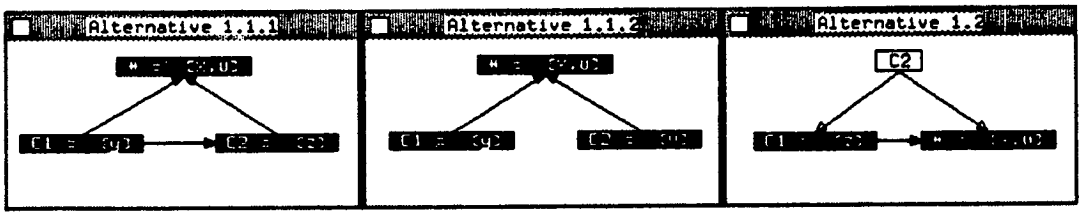
On dépile à présent l'alternative 1 et on expanse son premier noeud non-expansé. Celui-ci a pour contrainte associée C_1 et son ensemble de groupes de variables modifiables est $\{\{y\}, \{z\}\}$. On crée

[1] **Représentation des Alternatives :**
Les noeuds à expanser sont représentés en blanc et montrent le nom de la contrainte qui leur est associée. Les noeuds déjà expansés sont représentés en noir et montre le nom de leur contrainte ainsi que leurs conséquents c'est à dire les variables qu'ils recalculent. Les liens représentés figurent la relation d'antécédent entre les noeuds.

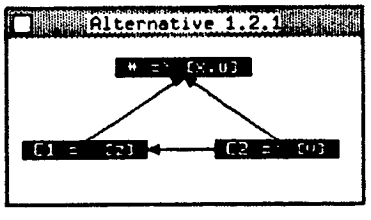
donc deux copies (alternatives 1.1 et 1.2) de l'alternative 1 auxquelles on associe respectivement $\{y\}$ et $\{z\}$ comme conséquents du noeud qu'on expanse. y n'intervient que dans C_1 et par conséquent, il n'y a ni lien, ni nouveau noeud à créer (C_1 n'est pas considérée puisque c'est la contrainte associée au noeud couramment expansé). En revanche, z intervient dans C_1 et C_2. Comme il existe déjà un noeud associé à C_2, on ajoute, en antécédent de ce noeud, le noeud expansé. Les deux copies sont successivement empilées et l'état de la pile est le suivant :



On procède ensuite de la même façon en dépilant l'alternative 1.1 et en considérant le noeud associé à C_2. Les groupes de variables modifiables sont ici $\{\{z\},\{v\}\}$. On donne ainsi naissance aux alternatives 1.1.1 et 1.1.2 et la pile devient :



1.1.1 est dépilée. C'est une alternative complète puisqu'elle ne comporte plus de noeuds à expanser. Il en va de même pour 1.1.2. On traite ensuite 1.2 en créant et en empilant 1.2.1 :



Enfin, 1.2.1 est dépilée et découverte complète. On trouve donc trois alternatives complètes (1.1.1, 1.1.2, 1.2.1) qui peuvent indifféremment être choisie pour restaurer la cohérence du réseau.

Bibliographie

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, 1974.

[Ba87] A. Borning and al. Constraint hierarchies. In *Proceedings of the OOPSLA 87*, 1987.

[Bat83] J. Batali. *Computational Introspection.* Technical Report 701, MIT AI Lab., 1983.

[Bor79] A. Borning. *ThingLab : A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, 1979.

[Cha87] J. Chailloux. *Le_Lisp Version 15.2.* 1987.

[CM84] W. Clocksin and C. Mellish. *Programming in Prolog,* 2^{nd} *ed.* Springer–Verlag, 1984.

[Cor88] O. Corby. *Un Tableau Réflexif pour la Coopération de Bases de Connaissances.* PhD thesis, Université de Nice, 1988.

[Cou78] *Cours Implémentation et Interprétation de Lisp.* Ecole de l'IRIA: Ecole de la Recherche, 1978.

[Dav87] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32:281–331, 1987.

[DCH87] R. Dieng, O. Corby, and P. Haren. Un système expert explicateur. In *Proceedings of COGNITIVA 87*, 1987.

[deK86a] J. deKleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[deK86b] J. deKleer. Extending the ATMS. *Artificial Intelligence*, 28:163–196, 1986.

[Din86] M. Dincbas. Constraints, logic programming and deductive databases. In *Proceedings of the France-Japan AI Symposium 86*, 1986.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[EG87] S. Ervin and M. Gross. ROADLAB - a constraint based laboratory for road design. In *Proceedings of the Second International Conference on Applications of AI to Engineering*, 1987.

[Fer87] J. Ferber. Reflection in computational systems. In *Proceedings of COGNITIVA 87*, 1987.

[FL69] G. Friedman and C. Leondes. Constraint theory, part I, II & III : fundamentals. *IEEE Transactions on Systems Science and Cybernetics*, 5(2), 1969.

[Fre78] E. Freuder. Synthesizing constraint expression. *Communications of the ACM*, 21(11), 1978.

[GJV87] H. Gusgen, U. Junker, and A. Voss. Constraints in a hybrid knowledge representation system. In *Proceedings of the IJCAI 87*, 1987.

[Gos83] J. Gosling. *Algebraic Constraints.* PhD thesis, Carnegie–Mellon University, 1983.

[Gus86] H. Gusgen. *Foundation of a System for Constraint Satisfaction:* CONSAT. Technical Report, GMD Sankt Augustin, 1986.

[Gus88] H. Gusgen. Some fundamental properties of local constraint propagation. *Artificial Intelligence*, 36:237–247, 1988.

[Har86] D. Harris. A hybrid structured object and constraint representation language. In *Proceedings of the AAAI 86*, 1986.

[KM77] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of the IFIP 77*, 1977.

[Lau86] J-L. Lauriere. *Intelligence Artificielle. Resolution de Problemes par, l'Homme et la Machine.* Editions Eyrolles, 1986.

[Mac77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mae86] P. Maes. *Reflection in an Object-Oriented Language.* Technical Report 86–8, Vrije Universiteit Brussel AI Lab., 1986.

[Mal87] J. Maleki. *ICONStraint : A Dependancy-Directed Constraint Maintenance System.* PhD thesis, Linkoping university, 1987.

[Per87] M. Perlin. On the computational equivalence of frame system and rule system. In *Proceedings of the US-Japan AI Symposium 87*, 1987.

[Ric83] E. Rich. *Artificial Intelligence, p. 176-184.* McGraw–Hill, 1983.

[Rit86] J-F. Rit. Propagating temporal constraints for scheduling. In *Proceedings of the AAAI 86*, 1986.

[SG87] D. Serrano and D. Gossard. Constraint management in conceptual design. In *Proceedings of the Second International Conference on Applications of AI to Engineering*, 1987.

[Sme88] *Smeci Version 1.3, Users' Reference Manual.* ILOG, 2 Av. Galliéni, F-94253 Gentilly, 1988.

[Smi82] B. Smith. *Reflection and Semantics in a Procedural Language.* Technical Report 272, MIT Laboratory for Computer Science., 1982.

[SS80] G. Sussman and G. Steele. CONSTRAINTS- a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[Ste80] G. Steele. *The Definition and Implementation of a Computer Programming Language Based on* CONSTRAINTS. PhD thesis, MIT, 1980.

[Ste87] S. Steel. On trying to do dependancy-directed backtracking by searching transformed state spaces (and failing). In *Proceedings of the AISB 87*, 1987.

[Sut63] I. Sutherland. *Sketchpad : A Man-Machine Graphical Communication System.* PhD thesis, MIT, 1963.

[Van87] K. VanMarcke. A parallel algorithm for consistency maintenance in knowledge representation. In B. DuBoulay, D. Hogg, and L. Steels, editors, *Advances in Artificial Intelligence - II*, North–Holland, 1987.

[Wal72] D. Waltz. *Generating semantic description from drawing of scenes with shadows.* Technical Report AI-271, MIT, 1972.

[ZMC87] R. Zabih, D. McAllester, and D. Chapman. Non-deterministic lisp with dependancy-directed backtracking. In *Proceedings of the AAAI 87*, 1987.

Table des matières

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique